(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
26 October 2006 (26.10.2006)

PCT

(10) International Publication Number
WO 2006/110957 A1

(51) International Patent Classification:
G06F 15/16 (2006.01)

(21) International Application Number:
PCT/AU2006/000532

(22) International Filing Date: 20 April 2006 (20.04.2006)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:

2005902023	21 April 2005 (21.04.2005)	AU
2005902024	21 April 2005 (21.04.2005)	AU
2005902025	21 April 2005 (21.04.2005)	AU
2005902026	21 April 2005 (21.04.2005)	AU
2005902027	21 April 2005 (21.04.2005)	AU

(71) Applicant *(for all designated States except US)*: WARATEK PTY LIMITED [AU/AU]; Suite 18, 12 Tyron Road, Lindfield, NSW 2070 (AU).

(72) Inventor; and
(75) Inventor/Applicant *(for US only)*: HOLT, John, Matthew [AU/AU]; Suite 18, 12 Tyron Road, Lindfield, NSW 2070 (AU).

(74) Agent: FRASER OLD & SOHN; Patent Attorneys, Level 6, 118 Alfred Steet, Milsons Point, NSW 2061 (AU).

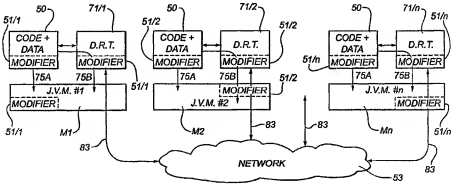
(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KN, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, LY, MA, MD, MG, MK, MN, MW, MX, MZ, NA, NG, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SM, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IS, IT, LT, LU, LV, MC, NL, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:
— *with international search report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: MODIFIED COMPUTER ARCHITECTURE FOR A COMPUTER TO OPERATE IN A MULTIPLE COMPUTER SYSTEM



(57) Abstract: A modified computer architecture (50, 71, #1, #2, #3) which enables applications program (50) to be run simultaneously on a plurality of computers (Ml, ...Mn) and a computer for the multiple computer system are disclosed. Shared memory at each computer is updated with amendments and/or overwrites so memory read requests are satisfied locally. During initial program loading (75) instructions which result in memory being re-written/manipulated are identified. Instructions are inserted to cause equivalent memory locations at all computers to be updated. Initialization of JAVA language classes and objects is disclosed so memory locations for all computers are initialized in the same manner. Finalization of JAVA language classes and objects is disclosed. Finalization occurs when the last class/object on all machines is no longer required. During initial program loading (75) instructions which result in the program (50) acquiring/releasing a lock on an asset (synchronization) are identified. Instructions are inserted to result in a modified synchronization routine with which all computers are updated. A single computer arranged to operate in a multiple computer system is disclosed.

WO 2006/110957 A1

MODIFIED COMPUTER ARCHITECTURE FOR A COMPUTER TO OPERATE IN A MULTIPLE COMPUTER SYSTEM

Field of the Invention

The present invention relates to computers and, in particular, to a modified machine architecture which enables the execution of different portions of an application program written to operate only on a single computer, substantially simultaneous on each of a plurality of computers interconnected via a communications network.

Background Art

Ever since the advent of computers, and computing, software for computers has been written to be operated upon a single machine. As indicated in Fig. 1, that single prior art machine 1 is made up from a central processing unit, or CPU, 2 which is connected to a memory 3 via a bus 4. Also connected to the bus 4 are various other functional units of the single machine 1 such as a screen 5, keyboard 6 and mouse 7.

A fundamental limit to the performance of the machine 1 is that the data to be manipulated by the CPU 2, and the results of those manipulations, must be moved by the bus 4. The bus 4 suffers from a number of problems including so called bus "queues" formed by units wishing to gain an access to the bus, conflict or -+contention problems, and the like. These problems can, to some extent, be alleviated by various stratagems including cache memory, however, such stratagems invariably increase the administrative overhead of the machine 1.

Naturally, over the years various attempts have been made to increase machine performance. One approach is to use symmetric multiple processors. This prior art approach has been used in so called "super" computers and is schematically indicated in Fig. 2. Here a plurality of CPU's 12 are connected to global memory 13. Again, a bottleneck arises in the communications between the CPU's 12 and the memory 13. This process has been termed "Single System Image". There is only one application and one whole copy of the memory for the application which is distributed over the global memory. The single application can read from and write to, (ie share) any memory location completely transparently.

Where there are a number of such machines interconnected via a network, this is achieved by taking the single application written for a single machine and partitioning the required memory resources into parts. These parts are then distributed across a number of computers to form the global memory 13 accessible by all CPU's 12. This procedure relies on masking, or hiding, the memory partition from the single running application program. The performance degrades when one CPU on one machine must access (via a network) a memory location physically located in a different machine.

Although super computers have been technically successful in achieving high computational rates, they are not commercially successful in that their inherent complexity makes them extremely expensive not only to manufacture but to administer. In particular, the single system image concept has never been able to scale over "commodity" (or mass produced) computers and networks. Specifically, the Single

System Image concept has only found practical application on very fast (and hence very expensive) computers interconnected by very fast (and similarly expensive) networks.

A further possibility of increased computer power through the use of a plural number of machines arises from the prior art concept of distributed computing which is schematically illustrated in Fig. 3. In this known arrangement, a single application program (Ap) is partitioned by its author (or another programmer who has become familiar with the application program) into various discrete tasks so as to run upon, say, three machines in which case "n" in Fig. 3 is the integer 3. The intention here is that each of the machines M1...M3 runs a different third of the entire application and the intention is that the loads applied to the various machines be approximately equal. The machines communicate via a network 14 which can be provided in various forms such as a communications link, the internet, intranets, local area networks, and the like. Typically the speed of operation of such networks 14 is an order of magnitude slower than the speed of operation of the bus 4 in each of the individual machines M1, M2, etc.

Distributed computing suffers from a number of disadvantages. Firstly, it is a difficult job to partition the application and this must be done manually. Secondly, communicating data, partial results, results and the like over the network 14 is an administrative overhead. Thirdly, the need for partitioning makes it extremely difficult to scale upwardly by utilising more machines since the application having been partitioned into, say three, does not run well upon four machines. Fourthly, in the event that one of the machines should become disabled, the overall performance of the entire system is substantially degraded.

A further prior art arrangement is known as network computing via "clusters" as is schematically illustrated in Fig. 4. In this approach, the entire application is loaded onto each of the machines M1, M2 ....Mn. Each machine communicates with a common database but does not communicate directly with the other machines. Although each machine runs the same application, each machine is doing a different "job" and uses only its own memory. This is somewhat analogous to a number of windows each of which sell train tickets to the public. This approach does operate, is scalable and mainly suffers from the disadvantage that it is difficult to administer the network.

In computer languages such as for example JAVA and MICROSOFT.NET there are two major types of constructs with which programmers deal. In the JAVA language these are known as objects and classes. More generally they may be referred to as assets. Every time an object (or other asset) is created there is an initialization routine run known as an object initialization (e.g., "<init>") routine. Similarly, every time a class is loaded there is a class initialization routine known as "<clinit>". Other languages use different terms but utilize a similar concept. In either case, however, there is no equivalent "clean up" or deletion routine to delete an object or class (or other asset) once it is no longer required. Instead, this "clean up" happens unobtrusively in a background mode.

Furthermore, in any computer environment it is necessary to acquire and release a lock to enable the use of such objects, classes, assets, resources or structures to avoid different parts of the application program from attempting to use the same objects, classes, assets, resources or structures at the one time. In the JAVA environment this is known as synchronization. Synchronization more generally refers to the exclusive use of an object, class, resource, structure, or other asset to avoid contention between and among computers or machines. This is achieved in JAVA by the "monitor enter" and "monitor exit" instructions or routines. Other languages use different terms but utilize a similar concept.

Unfortunately, conventional computing systems, architectures, and operating schemes do not provide for computing environments and methods in which an application program can operate simultaneously on an arbitrary plurality of computers where the environment and operating scheme ensure that the abovementioned memory management, initialization, clean up and synchronization procedures operate in a consistent and coordinated fashion across all the computing machines.

The genesis of the present invention is a desire to provide a multiple computer system (and related arrangements such as individual computers which can operate in such a system, and a method of operating such computers) which to some extent ameliorates the problems of prior art multiple computer systems.

SUMMARY OF THE INVENTION

The present invention discloses a computing environment in which an application program operates simultaneously on a plurality of computers. In such an environment it is advantageous to ensure that the abovementioned asset initialization, clean-up and synchronization procedures operate in a consistent and coordinated fashion across all the machines.

In accordance with a first aspect of the present invention there is disclosed a single computer intended to operate in a multiple computer system which comprises a plurality of computers each having a local memory and each being interconnected via a communications network, wherein a different portion of at least one application program each written to execute on only a single computer executes substantially simultaneously on a corresponding one of said plurality of computers, and at least one memory location is replicated in the local memory of each said computer, said single computer comprising:
a local memory having at least one memory location intended to be updated via said communications network,
a communications port for connection to said communications network, and
updating means to transfer to said communications port any updated content(s) of said replicated local memory location(s) whereby the corresponding replicated memory location of each said computer of said multiple system can be updated via said communicating network and all said replicated memory locations can remain substantially identical.

In accordance with a second aspect of the present invention there is disclosed a single computer intended to operate in a multiple computer system which comprises a plurality of computers each having a local memory and each being interconnected via a communications network, wherein a different portion of at least one application program each written to execute on only a single computer executes substantially simultaneously on a corresponding one of said plurality of computers, and at least one memory location is replicated in the local memory of each said computer, said single computer comprising:

a local memory having at least one memory location intended to be updated via said communications network,
a communications port for connection to said communications network,
updating means to transfer to said communications port any updated content(s) of said replicated local memory location(s), and
initialization means which determine the initial content or value of said replicated memory location and which can be disabled.

In accordance with a third aspect of the present invention there is disclosed a A single computer intended to operate in a multiple computer system which comprises a plurality of computers each having a local memory and each being interconnected via a communications network, wherein a different portion of at least one application program each written to execute on only a single computer executes substantially simultaneously on a corresponding one of said plurality of computers, and at least one memory location is replicated in the local memory of each said computer, said single computer comprising:
a local memory having at least one memory location intended to be updated via said communications network,
a communications port for connection to said communications network,
updating means to transfer to said communications port any updated content(s) of said replicated local memory location(s), and
finalization means which deletes said replicated memory location when all said computers no longer need to refer thereto, said finalization means being connected to said communications port to receive therefrom data transmitted over said network relating to continued reference of other computers of said multiple computer system to said replicated memory location.

In accordance with a fourth aspect of the present invention there is disclosed a A single computer intended to operate in a multiple computer system which comprises a plurality of computers each having a local memory and each being interconnected via a communications network, wherein a different portion of at least one application program each written to execute on only a single computer executes substantially simultaneously on a corresponding one of said plurality of computers, and at least one memory location is replicated in the local memory of each said computer, said single computer comprising:
a local memory having at least one memory location intended to be updated via said communications network,
a communications port for connection to said communications network,
updating means to transfer to said communications port any updated content(s) of said replicated local memory location(s), and
lock acquisition and relinquishing means to respectively permit said replicated local memory location to be written to, and prevent said replicated local memory being written to, on command.

In accordance with a fifth aspect of the present invention there is disclosed a single computer intended to operate in a multiple computer system which comprises a plurality of computers each having a local memory and each being interconnected via a communications network, wherein a different portion of at least one application program each written to execute on only a single computer executes substantially simultaneously on a corresponding one of said plurality of computers, and at least one memory location

is replicated in the local memory of each said computer, said single computer comprising:
a local memory having at least one memory location intended to be updated via said communications network,
a communications port for connection to said communications network,
updating means to transfer to said communications port any updated content(s) of said replicated local memory location(s) whereby the corresponding replicated memory location of each said computer of said multiple system can be updated via said communicating network and all said replicated memory locations can remain substantially identical,
initialization means which determine the initial content or value of said replicated memory location and which can be disabled,
finalization means which deletes said replicated memory location when all said computers no longer need to refer thereto, said finalization means being connected to said communications port to receive therefrom data transmitted over said network relating to continued reference of other computers of said multiple computer system to said replicated memory location, and
lock acquisition and relinquishing means to respectively permit said replicated local memory location to be written to, and prevent said replicated local memory being written to, on command.

In accordance with a sixth aspect of the present invention there is disclosed a multiple computer system having at least one application program each written to operate on only a single computer but running simultaneously on a plurality of computers interconnected by a communications network, wherein different portions of said application program(s) execute substantially simultaneously on different ones of said computers, wherein each computer has an independent local memory accessible only by the corresponding portion of said application program(s) and wherein for each said portion a like plurality of substantially identical objects are created, each in the corresponding computer.

In accordance with a seventh aspect of the present invention there is disclosed a plurality of computers interconnected via a communications link and each having an independent local memory and substantially simultaneously operating a different portion at least one application program each written to operate on only a single computer, each local memory being accessible only by the corresponding portion of said application program.

In accordance with a eighth aspect of the present invention there is disclosed a multiple computer system having at least one application program each written to operate on only a single computer but running substantially simultaneously on a plurality of computers interconnected by a communications network, wherein different portions of said application program(s) execute substantially simultaneously on different ones of said computers and for each said portion a like plurality of substantially identical objects are created, each in the corresponding computer and each having a substantially identical name, and wherein the initial contents of each of said identically named objects is substantially the same.

In accordance with a ninth aspect of the present invention there is disclosed a plurality of computers interconnected via a communications link and substantially simultaneously operating at least one application program each written to operation on only a single computer wherein each said computer substantially simultaneously executes a different portion of said application program(s), each said computer in operating its application program portion creates objects only in local memory physically located in each said computer, the contents of the local memory utilized by each said computer are fundamentally similar but not, at each instant, identical, and every one of said computers has distribution update means to distribute to all other said computers objects created by said one computer.

In accordance with a tenth aspect of the present invention there is disclosed a multiple computer system having at least one application program each written to operate only on a single computer but running substantially simultaneously on a plurality of computers interconnected by a communications network, wherein different portions of said application program(s) execute substantially simultaneously on different ones of said computers and for each said portion a like plurality of substantially identical objects are created, each in the corresponding computer and each having a substantially identical name, and wherein all said identical objects are collectively deleted when each one of said plurality of computers no longer needs to refer to their corresponding object.

In accordance with an eleventh aspect of the present invention there is disclosed a plurality of computers interconnected via a communications link and operating substantially simultaneously at least one application program each written to operate only on a single computer, wherein each said computer substantially simultaneously executes a different portion of said application program(s), each said computer in operating its application program portion needs, or no longer needs to refer to an object only in local memory physically located in each said computer, the contents of the local memory utilized by each said computer is fundamentally similar but not, at each instant, identical, and every one of said computers has a finalization routine which deletes a non-referenced object only if each one of said plurality of computers no longer needs to refer to their corresponding object.

In accordance with a twelfth aspect of the present invention there is disclosed a multiple computer system having at least one application program each written to operate on only a single computer but running substantially simultaneously on a plurality of computers interconnected by a communications network, wherein different portions of said application program(s) execute substantially simultaneously on different ones of said computers and for each portion a like plurality of substantially identical objects are created, each in the corresponding computer and each having a substantially identical name, and said system including a lock means applicable to all said computers wherein any computer wishing to utilize a named object therein acquires an authorizing lock from said lock means which permits said utilization and which prevents all the other computers from utilizing their corresponding named object until said authorizing lock is relinquished.

In accordance with a thirteenth aspect of the present invention there is disclosed a plurality of computers interconnected via a communications link and operating substantially simultaneously at least one application program each written to operate on only a single computer, wherein each said computer substantially simultaneously executes a different portion of said application program(s), each said computer in operating its application program portion utilizes an object only in local memory physically located in each said computer, the contents of the local memory utilized by each said computer is fundamentally similar but not, at each instant, identical, and every one of said computers has an acquire lock routine and a release lock routine which permit utilization of the local object only by one computer and each of the remainder of said plurality of computers is locked out of utilization of their corresponding object.

In accordance with a fourteenth aspect of the present invention there is disclosed a method of running simultaneously on a plurality of computers at least one application program each written to operate on only a single computer, said computers being interconnected by means of a communications network, said method comprising the step of,
(i) executing different portions of said application program(s) on different ones of said computers and for each said portion creating a like plurality of substantially identical objects each in the corresponding computer and each accessible only by the corresponding portion of said application program.

In accordance with a fifteenth aspect of the present invention there is disclosed a method of loading an application program written to operate only on a single computer onto each of a plurality of computers, the computers being interconnected via a communications link, and different portions of said application program(s) being substantially simultaneously executable on different computers with each computer having an independent local memory accessible only by the corresponding portion of said application program(s), the method comprising the step of modifying the application before, during, or after loading and before execution of the relevant portion of the application program.

In accordance with a sixteenth aspect of the present invention there is disclosed a method of operating simultaneously on a plurality of computers all interconnected via a communications link at least one application program each written to operate on only a single computer, each of said computers having at least a minimum predetermined local memory capacity, different portions of said application program(s) being substantially simultaneously executed on different ones of said computers with the local memory of each computer being only accessible by the corresponding portion of said application program(s), said method comprising the steps of:
(i) initially providing each local memory in substantially identical condition,
(ii) satisfying all memory reads and writes generated by each said application program portion from said corresponding local memory, and
(iii) communicating via said communications link all said memory writes at each said computer which take place locally to all the remainder of said plurality of computers whereby the contents of the local memory utilised by each said computer, subject to an updating data transmission delay, remains substantially identical.

In accordance with a seventeenth aspect of the present invention there is disclosed a method of compiling or modifying an application program written to operate on only a

single computer but to run simultaneously on a plurality of computers interconnected via a communications link, with different portions of said application program(s) executing substantially simultaneously on different ones of said computers each of which has an independent local memory accessible only by the corresponding portion of said application program, said method comprising the steps of:
(i) detecting instructions which share memory records utilizing one of said computers,
(ii) listing all such shared memory records and providing a naming tag for each listed memory record,
(iii) detecting those instructions which write to, or manipulate the contents of, any of said listed memory records, and
(iv) activating an updating propagation routine following each said detected write or manipulate instruction, said updating propagation routine forwarding the re-written or manipulated contents and name tag of each said re-written or manipulated listed memory record to the remainder of said computers.

In accordance with an eighteenth aspect of the present invention there is disclosed a multiple thread processing computer operation in which individual threads of a single application program written to operate on only a single computer are simultaneously being processed each on a different corresponding one of a plurality of computers each having an independent local memory accessible only by the corresponding thread and each being interconnected via a communications link, the improvement comprising communicating changes in the contents of local memory physically associated with the computer processing each thread to the local memory of each other said computer via said communications link.

In accordance with a nineteenth aspect of the present invention there is disclosed a method of running substantially simultaneously on a plurality of computers at least one application program each written to operate on only a single computer, said computers being interconnected by means of a communications network, said method comprising the steps of:
(i) executing different portions of said application program(s) on different ones of said computers and for each said portion creating a like plurality of substantially identical objects each in the corresponding computer and each having a substantially identical name, and
(ii) creating the initial contents of each of said identically named objects substantially the same.

In accordance with a twentieth aspect of the present invention there is disclosed a method of compiling or modifying an application program written to operate on only a single computer to have different portions thereof to execute substantially simultaneously on different ones of a plurality of computers interconnected via a communications link, said method comprising the steps of:
(i) detecting instructions which create objects utilizing one of said computers,
(ii) activating an initialization routine following each said detected object creation instruction, said initialization routine forwarding each created object to the remainder of said computers.

In accordance with a twenty first aspect of the present invention there is disclosed a multiple thread processing computer operation in which individual threads of a single application program written to operate on only a single computer are substantially simultaneously being processed each on a different corresponding one of a plurality of computers interconnected via a communications link, the improvement comprising communicating objects created in local memory physically associated with the computer processing each thread to the local memory of each other said computer via said communications link.

In accordance with a twenty second aspect of the present invention there is disclosed a method of ensuring consistent initialization of an application program written to operate on only a single computer but different portions of which are to be executed substantially simultaneously each on a different one of a plurality of computers interconnected via a communications network, said method comprising the steps of:
(i) scrutinizing said application program at, or prior to, or after loading to detect each program step defining an initialization routine, and
(ii) modifying said initialization routine to ensure consistent operation of all said computers.

In accordance with a twenty third aspect of the present invention there is disclosed a method of running substantially simultaneously on a plurality of computers at least one application program each written to operate only on a single computer, said computers being interconnected by means of a communications network, said method comprising the steps of:
(i) executing different portions of said application program(s) on different ones of said computers and for each said portion creating a like plurality of substantially identical objects each in the corresponding computer and each having a substantially identical name, and
(ii) deleting all said identical objects collectively when all of said plurality of computers no longer need to refer to their corresponding object.

In accordance with a twenty fourth aspect of the present invention there is disclosed a method of ensuring consistent finalization of an application program written to operate only on a single computer but different portions of which are to be executed substantially simultaneously each on a different one of a plurality of computers interconnected via a communications network, said method comprising the steps of:
(i) scrutinizing said application program at, or prior to, or after loading to detect each program step defining an finalization routine, and
(ii) modifying said finalization routine to ensure collective deletion of corresponding objects in all said computers only when each one of said computers no longer needs to refer to their corresponding object.

In accordance with a twenty fifth aspect of the present invention there is disclosed a multiple thread processing computer operation in which individual threads of a single application program written to operate only on a single computer are substantially simultaneously being processed each on a corresponding different one of a plurality of computers interconnected via a communications link, and in which objects in local memory physically associated with the computer processing each thread have corresponding objects in the local memory of each other said computer, the improvement comprising collectively deleting all said corresponding objects when each one of said plurality of computers no longer needs to refer to their corresponding object.

In accordance with a twenty sixth aspect of the present invention there is disclosed a method of running substantially simultaneously on a plurality of computers at least one application program each written to operate only on a single computer, said computers being interconnected by means of a communications network, said method comprising the steps of:

(i) executing different portions of said application program(s) on different ones of said computers and for each said portion creating a like plurality of substantially identical objects each in the corresponding computer and each having a substantially identical name, and

(ii) requiring any of said computers wishing to utilize a named object therein to acquire an authorizing lock which permits said utilization and which prevents all the other computers from utilizing their corresponding named object until said authorizing lock is relinquished.

In accordance with a twenty seventh aspect of the present invention there is disclosed a method of ensuring consistent synchronization of an application program written to operate only on a single computer but different portions of which are to be executed substantially simultaneously each on a different one of a plurality of computers interconnected via a communications network, said method comprising the steps of:

(i) scrutinizing said application program at, or prior to, or after loading to detect each program step defining an synchronization routine, and

(ii) modifying said synchronization routine to ensure utilization of an object by only one computer and preventing all the remaining computers from simultaneously utilizing their corresponding objects.

In accordance with a twenty eighth aspect of the present invention there is disclosed a multiple thread processing computer operation in which individual threads of a single application program written to operate only on a single computer are substantially simultaneously being processed each on a corresponding different one of a plurality of computers interconnected via a communications link, and in which objects in local memory physically associated with the computer processing each thread have corresponding objects in the local memory of each other said computer, the improvement comprising permitting only one of said computers to utilize an object and preventing all the remaining computers from simultaneously utilizing their corresponding object.

In accordance with a twenty ninth aspect of the present invention there is disclosed a computer program product comprising a set of program instructions stored in a storage medium and operable to permit one or a plurality of computers to carry out the abovementioned methods.

In accordance with a thirtieth aspect of the invention there is disclosed a distributed run time and distributed run time system adapted to enable communications between a plurality of computers, computing machines, or information appliances.

In accordance with a thirty first aspect of the invention there is disclosed a modifier, modifier means, and modifier routine for modifying an application program written to execute on a single computer or computing machine whereby the modified application program executes substantially simultaneously on a plurality of networked computers or computing machines.

In accordance with a thirty second aspect of the present invention there is disclosed a computer program and computer program product written to operate on only a single computer but product comprising a set of program instructions stored in a storage medium and operable to permit a plurality of computers to carry out the abovementioned procedures, routines, and methods.

Brief Description of the Drawings

Embodiments of the present invention will now be described with reference to the drawings in which:

Fig. 1 is a schematic view of the internal architecture of a conventional computer,

Fig. 2 is a schematic illustration showing the internal architecture of known symmetric multiple processors,

Fig. 3 is a schematic representation of prior art distributed computing,

Fig. 4 is a schematic representation of a prior art network computing using clusters,

Fig. 5 is a schematic block diagram of a plurality of machines operating the same application program in accordance with a first embodiment of the present invention,

Fig. 6 is a schematic illustration of a prior art computer arranged to operate JAVA code and thereby constitute a JAVA virtual machine,

Fig. 7 is a drawing similar to Fig. 6 but illustrating the initial loading of code in accordance with the preferred embodiment,

Fig. 8 is a drawing similar to Fig. 5 but illustrating the interconnection of a plurality of computers each operating JAVA code in the manner illustrated in Fig. 7,

Fig. 9 is a flow chart of the procedure followed during loading of the same application on each machine in the network,

Fig. 10 is a flow chart showing a modified procedure similar to that of Fig. 9,

Fig. 11 is a schematic representation of multiple thread processing carried out on the machines of Fig. 8 utilizing a first embodiment of memory updating,

Fig. 12 is a schematic representation similar to Fig. 11 but illustrating an alternative embodiment,

Fig. 13 illustrates multi-thread memory updating for the computers of Fig. 8,

Fig. 14 is a schematic illustration of a prior art computer arranged to operate in JAVA code and thereby constitute a JAVA virtual machine,

Fig. 15 is a schematic representation of n machines running the application program and serviced by an additional server machine X,

Fig. 16 is a flow chart of illustrating the modification of initialization routines,

Fig. 17 is a flow chart illustrating the continuation or abortion of initialization routines,

Fig. 18 is a flow chart illustrating the enquiry sent to the server machine X,

Fig. 19 is a flow chart of the response of the server machine X to the request of Fig. 18,

Fig. 20 is a flowchart illustrating a modified initialization routine for the <clinit>

instruction,

Fig. 21 is a flowchart illustrating a modified initialization routine for the <init> instruction,

Fig. 22 is a flow chart of illustrating the modification of "clean up" or finalization routines,

Fig. 23 is a flow chart illustrating the continuation or abortion of finalization routines,

Fig. 24 is a flow chart illustrating the enquiry sent to the server machine X,

Fig. 25 is a flow chart of the response of the server machine X to the request of Fig. 24,

Fig. 26 is a flow chart of illustrating the modification of the monitor enter and exit routines,

Fig. 27 is a flow chart illustrating the process followed by processing machine in requesting the acquisition of a lock,

Fig. 28 is a flow chart illustrating the requesting of the release of a lock,

Fig. 29 is a flow chart of the response of the server machine X to the request of Fig. 27,

Fig. 30 is a flow chart illustrating the response of the server machine X to the request of Fig. 28,

Fig. 31 is a schematic representation of two laptop computers interconnected to simultaneously run a plurality of applications, with both applications running on a single computer,

Fig. 32 is a view similar to Fig. 31 but showing the Fig. 31 apparatus with one application operating on each computer, and

Fig. 33 is a view similar to Figs. 31 and 32 but showing the Fig. 31 apparatus with both applications operating simultaneously on both computers.

REFERENCE TO ANNEXES

Although the specification provides a complete and detailed description of the several embodiments of the invention such that the invention may be understood and implemented without reference to other materials, the specification does includes Annexures A, B, C and D which provide exemplary actual program or code fragments which implement various aspects of the described embodiments. Although aspects of the invention are described throughout the specification including the Annexes, drawings, and claims, it may be appreciated that Annexure A relates primarily to fields, Annexure B relates primarily to initialization, Annexure C relates primarily to finalization, and Annexure D relates primarily to synchronization. More particularly, the accompanying Annexures are provided in which:

Annexures A1-A10 illustrate exemplary code to illustrate embodiments of the invention in relation to fields.

Annexure B1 is an exemplary typical code fragment from an unmodified class initialization <clinit> instruction, Annexure B2 is an equivalent in respect of a modified class initialization <clinit> instruction. Annexure B3 is a typical code fragment from an unmodified object initialization <init> instruction. Annexure B4 is an equivalent in respect of a modified object initialization <init> instruction. In addition, Annexure B5 is an alternative to the code of Annexure B2 for an unmodified class initialization instruction, and Annexure B6 is an alternative to the code of Annexure B4 for a modified object initialization <init> instruction. Furthermore, Annexure B7 is exemplary computer program source-code of InitClient, which queries an "initialization server" for the initialization status of the relevant class or object. Annexure B8 is the computer program source-code of InitServer, which receives an initialization status query by InitClient and in response returns the corresponding status. Similarly, Annexure B9 is the computer program source-code of the example application used in the before/after examples of Annexure B1-B6.

It will be appreciated in light of the description provided here that the categorization of the Annexures as well as the use of other headings and subheadings in this description is intended as an aid to the reader and is not to be used to limit the scope of the invention in any way.

Detailed Description

The present invention discloses a modified computer architecture which enables an applications program to be run simultaneously on a plurality of computers in a manner that overcomes the limitations of the aforedescribed conventional architectures, systems, methods, and computer programs.

In one aspect, shared memory at each computer may be updated with amendments and/or overwrites so that all memory read requests are satisfied locally. Before, during or after program loading, but before execution of relevant portions of the program code are executed, or similar, instructions which result in memory being re-written or manipulated are identified. Additional instructions are inserted into the program code (or other modification made) to cause the equivalent memory locations at all computers to be updated. While the invention is not limited to JAVA language or virtual machines, exemplary embodiments are described relative to the JAVA language and standards. In another aspect, the initialization of JAVA language classes and objects (or other assets) are provided for so all memory locations for all computers are initialized in the same manner. In another aspect, the finalization of JAVA language classes and objects is also provide so finalization only occurs when the last class or object present on all machines is no longer required. In still another aspect, synchronization is provided such that instructions which result in the application program acquiring (or releasing) a lock on a particular asset (synchronization) are identified. Additional instructions are inserted (or other code modifications performed) to result in a modified synchronization routine with which all computers are updated.

The present invention also discloses a computing environment and computing method in which an application program operates simultaneously on a plurality of computers. In such an environment it is advantageous to ensure that the abovementioned initialization, clean-up and synchronization procedures operate in a consistent and coordinated fashion across all the machines. These memory replication, object or other asset initialization, finalization, and synchronization may be used and applied separately in a variety of computing and information processing environments. Furthermore, they may advantageously be implemented and applied in any combination so as to provide synergistic effects for multi-computer processing, such as network based distributed computing.

As each of the architectural, system, procedural, method and computer program aspects of the invention (e.g., memory management and replication, initialization, finalization, and synchronization) may be applied separately, they are thus first described without specific reference to the other aspects. It will however be appreciated in light of the descriptions provided that the object, class, or other asset creation or initialization may generally precede finalization of such objects, classes, or other assets.

In addition, during the loading of, or at any time preceding the execution of, the application code 50 (or relevant portion thereof) on each machine M1, M2...Mn, each application code 50 has been modified by the corresponding modifier 51 according to the same rules (or substantially the same rules since minor optimizing changes are permitted within each modifier 51/1, 51/2, ..., 51/n). Where separate modifications are required on any particular machine, such as to machine M2, to effect the memory management, initialization, finalization, and/or synchronization for that machine, then each machine may in fact have and be modified according to a plurality of separate modifiers (such as 51/2-M (e.g., M2 memory management modifier), 51/2-I (e.g., M2 initialization modifier), 51/2-F (e.g., M2 finalization modifier), and/or 51/2-S (e.g., M2 synchronization modifier); or alternatively any one or more of these modifiers may be combined into a combined modifier for that computer or machine. In at least some embodiments, efficiencies will result from performing the steps required to identify the modification required, in performing the actual modification, and in coordinating the operation of the plurality or constellation of computers or machines in an organized, consistent, and coherent manner. These modifications may be performed in accordance with aspects of the invention by the distributed run time means 71 described in greater detail hereinafter. In analogous manner those workers having ordinary skill in the art in light of the description provided herein will appreciate that the structural and methodological aspects of the distributed run time, distributed run time system, and distributed run time means as they are described herein specifically to memory management, initialization, finalization, and/or synchronization may be combined so any of the modifications required to an application program or code may be made separately or in combination to achieve any required memory management, initialization, finalization, and/or synchronization on any particular machine and across the plurality of machines M1, M2, ..., Mn.

With specific reference to any memory management modifier that may be provided, such memory management modifier 51-M or DRT 71-M or other code modifying means component of the overall modifier or distributed run time means is responsible for creating or replicating a memory structure and contents on each of the individual machines M1, M2...Mn that permits the plurality of machines to interoperate. In some embodiments this replicated memory structure will be identical, in other embodiments this memory structure will have portions that are identical and other portions that are not, and in still other embodiments the memory structures are or may not be identical.

With reference to any initialisation modifier that may be present, such initialisation modifier 51-I or DRT 71-I or other code modifying means component of the overall modifier or distributed run time means is responsible for modifying the application code 50 so that it may execute initialisation routines or other initialization operations, such as for example class and object initialization methods or routines in the JAVA language and virtual machine environment, in a coordinated, coherent, and consistent manner across the plurality of individual machines M1, M2...Mn.

With reference to the finalization modifier that may be present, such finalization modifier 51-F or DRT 71-F or other code modifying means is responsible for modifying the application code 50 so that the code may execute finalization clean-up, or other memory reclamation, recycling, deletion or finalization operations, such as for example finalization methods in the JAVA language and virtual machine environment, in a coordinated, coherent and consistent manner across the plurality of individual machines M1, M2, ..., Mn.

Furthermore, with reference to any synchronization modifier that may be present, such synchronization modifier 51-S or DRT 71-S or other code modifying means is responsible for ensuring that when a part (such as a thread or process) of the modified application program 50 running on one or more of the machines exclusively utilizes (e.g., by means of a synchronization routine or similar or equivalent mutual exclusion operator or operation) a particular local asset, such as an objects 50X-50Z or class 50A, no other different and potentially concurrently executing part on machines M2...Mn exclusively utilizes the similar equivalent corresponding asset in its local memory at once or at the same time.

These structures and procedures when applied in combination when required, maintain a computing environment where memory locations, address ranges, objects, classes, assets, resources, or any other procedural or structural aspect of a computer or computing environment are where required created, maintained, operated, and deactivated or deleted in a coordinated, coherent, and consistent manner across the plurality of individual machines M1, M2...Mn.

The embodiments will be described with reference to the JAVA language, however, it will be apparent to those skilled in the art that the invention is not limited to this language and, in particular can be used with the similar languages (including procedural, declarative and object oriented languages) including the MICROSOFT.NET platform and architecture (Visual Basic, Visual C, and Visual C++, and Visual C#), FORTRAN, C, C++, COBOL, BASIC and the like.

In connection with Fig. 5, in accordance with a preferred embodiment of the present invention a single application program 50 can be operated simultaneously on a number of computers or machines M1, M2...Mn communicating via network 53. As it will become apparent hereafter, each of the machines M1, M2...Mn operates with the same application program 50 on each machine M1, M2...Mn and thus all of the machines M1, M2...Mn have the same, or substantially the same, application code and data 50. Similarly, each of the machines M1, M2...Mn operates with the same (or substantially the same) modifier 51 on each machine M1, M2...Mn and thus all of the machines M1, M2...Mn have the same (or substantially the same) modifier 51 with the modifier of machine M2 being designated 51/2. In addition, during the loading of, or preceding the execution of, the application 50 on each machine M1, M2...Mn, each application 50 has been modified by the corresponding modifier 51 according to the same rules (or substantially the same rules since minor optimising changes are permitted within each modifier 51/1 ...51/n).

As a consequence of the above described arrangement, if each of the machines M1, M2...Mn has, say, a shared memory capability of 10MB, then the total shared memory available to each application 50 is not, as one might expect, 10n MB. However, how this results in improved operation will become apparent hereafter. Naturally, each machine M1, M2...Mn has an unshared memory capability. The unshared memory capability of the machines M1, M2...Mn are normally approximately equal but need not be.

It is known in the prior art to provide a single computer or machine (produced by any one of various manufacturers and having an operating system operating in any one of various different languages) utilizing the particular language of the application by creating a virtual machine as illustrated in Fig 6.

The code and data and virtual machine configuration or arrangement of Fig 6 takes the form of the application code 50 written in the JAVA language and executing within the JAVA virtual machine 61. Thus where the intended language of the application is the language JAVA, a JAVA virtual machine is used which is able to operate code in JAVA irrespective of the machine manufacturer and internal details of the computer or machine.

For further details, see "The JAVA Virtual Machine Specification" 2nd Edition by T. Lindholm and F. Yellin of Sun Microsystems Inc of the USA which is incorporated by reference herein.

This conventional art arrangement of Fig 6 is modified in accordance with embodiments of the present invention by the provision of an additional facility which is conveniently termed a "distributed run time" or a "distributed run time system" DRT 71 and as seen in Fig 7.

In Figs. 7 and 8, the application code 50 is loaded onto the Java Virtual Machine(s) M1, M2,...Mn in cooperation with the distributed runtime system 71, through the loading procedure indicated by arrow 75 or 75A or 75B. As used herein the terms "distributed runtime" and the "distributed run time system" are essentially synonymous, and by means of illustration but not limitation are generally understood to include library code and processes which support software written in a particular language running on a particular platform. Additionally, a distributed runtime system may also include library code and processes which support software written in a particular language running within a particular distributed computing environment. The runtime system typically deals with the details of the interface between the program and the operating system such as system calls, program start-up and termination, and memory management. For purposes of background, a conventional Distributed Computing Environment (DCE) (that does not provide the capabilities of the inventive distributed run time or distributed run time system 71 used in the preferred embodiments of the present invention) is available from the Open Software Foundation. This Distributed Computing Environment (DCE) performs a form of computer-to-computer communication for software running on the machines, but among its many limitations, it is not able to implement the desired modification or communication operations. Among its functions and operations the preferred DRT 71 coordinates the particular communications between the plurality of machines M1, M2,...Mn. Moreover, the preferred distributed runtime 71 comes into operation during the loading procedure indicated by arrow 75A or 75B of the JAVA application 50 on each JAVA virtual machine 72 or machines JVM#1, JVKMJ#2,...JVM#n of Fig. 8. It will be appreciated in light of the description provided herein that although many examples and descriptions are provided relative to the JAVA language and JAVA virtual machines so that the reader may get the benefit of specific examples, the invention is not restricted to either the JAVA language or JAVA virtual machines, or to any other language, virtual machine, machine or operating environment.

Fig 8 shows in modified form the arrangement of the JAVA virtual machines, each as illustrated in Fig 7. It will be apparent that again the same application code 50 is loaded onto each machine M1, M2...Mn. However, the communications between each machine M1, M2...Mn are as indicated by arrows 83, and although physically routed through the machine hardware, are advantageously controlled by the individual DRT's 71/1...71/n within each machine. Thus, in practice this may be conceptionalised as the DRT's 71/1,...71/n communicating with each other via the network or other communications link 53 rather than the machines M1, M2...Mn communicating directly themselves or with each other. Contemplated and included are either this direct communication between machines M1, M2...Mn or DRT's 71/1, 71/2...71/n or a combination of such communications. The preferred DRT 71 provides communication that is transport, protocol, and link independent.

The one common application program or application code 50 and its executable version (with likely modification) is simultaneously or concurrently executing across the plurality of computers or machines M1, M2...Mn. The common application program 5. The application program 5 is written with the intention that it only operate on a single machine or computer. Essentially the modified structure is to replicate and identical memory structure and contents on each of the individual machines

The term common application program is to be understood to mean an application program or application program code written to operate on a single machine, and loaded and/or executed in whole or in part on each one of the plurality of computers or machines M1, M2...Mn, or optionally on each one of some subset of the plurality of computers or machines M1, M2...Mn. Put somewhat differently, there is a common application program represented in application code 50. This is either a single copy or a plurality of identical copies each individually modified to generate a modified copy or version of the application program or program code. Each copy or instance is then prepared for execution on the corresponding machine. At the point after they are modified they are common in the sense that they perform similar operations and operate consistently and coherently with each other. It will be appreciated that a plurality of computers, machines, information appliances, or the like implementing embodiments of the invention may optionally be connected to or coupled with other computers, machines, information appliances, or the like that do not implement embodiments of the invention.

The same application program 50 (such as for example a parallel merge sort, or a computational fluid dynamics application or a data mining application)is run on each machine, but the executable code of that application program is modified on each machine as necessary such that each executing instance (copy or replica) on each machine coordinates its local operations on that particular machine with the operations of the respective instances (or copies or replicas) on the other machines such that they function together in a consistent, coherent and coordinated manner and give the appearance of being one global instance of the application (i.e. a "meta-application").

The copies or replicas of the same or substantially the same application codes, are each loaded onto a corresponding one of the interoperating and connected machines or computers. As the characteristics of each machine or computer may differ, the application code 50 may be modified before loading, during the loading process, and with some disadvantages after the loading process, to provide a customization or modification of the code on each machine. Some dissimilarity between the programs may be permitted so long as the other requirements for interoperability, consistency, and coherency as described herein can be maintained. As it will become apparent hereafter, each of the machines M1, M2...Mn and thus all of the machines M1, M2...Mn have the same or substantially the same application code 50, usually with a modification that may be machine specific.

Before the loading of, during the loading of, or at any time preceding the execution of, the application code 50 (or the relevant portion thereof) on each machine M1, M2...Mn, each application code 50 is modified by a corresponding modifier 51 according to the same rules (or substantially the same rules since minor optimizing changes are permitted within each modifier 51/1, 51/2...51/n).

Each of the machines M1, M2...Mn operates with the same (or substantially the same or similar) modifier 51 (in some embodiments implemented as a distributed run time or DRT71 and in other embodiments implemented as an adjunct to the code and data 50, and also able to be implemented either to the JAVA virtual machine itself). Thus all of the machines M1, M2...Mn have the same (or substantially the same or similar) modifier 51 for each modification required. A different modification, for example, may be required for memory management and replication, for initialization, for finalization, and/or for synchronization (though not all of these modification types may be required for all embodiments).

There are alternative implementations of the modifier 51 and the distributed run time 71. For example as indicated by broken lines in Fig. 8, the modifier 51 may be implemented as a component of or within the distributed run time 71, and therefore the DRT 71 may implement the functions and operations of the modifier 51. Alternatively, the function and operation of the modifier 51 may be implemented outside of the structure, software, firmware, or other means used to implement the DRT 71 such as within the code and data 50, or within the JAVA virtual machine itself. In one embodiment, both the modifier 51 and DRT 71 are implemented or written in a single piece of computer program code that provides the functions of the DRT and modifier. In this case the modifier function and structure is, in practice, subsumed into the DRT. Independent of how it is implemented, the modifier function and structure is responsible for modifying the executable code of the application code program, and the distributed run time function and structure is responsible for implementing communications between and among the computers or machines. The communications functionality in one embodiment is implemented via an intermediary protocol layer within the computer program code of the DRT on each machine. The DRT can, for example, implement a communications stack in the JAVA language and use the Transmission Control Protocol/Internet Protocol (TCP/IP) to provide for communications or talking between the machines. Exactly how these functions or operations are implemented or divided between structural and/or procedural elements, or between computer program code or data structures, is not crucial.

However, in the arrangement illustrated in Fig 8, a plurality of individual computers or machines M1, M2...Mn are provided, each of which are interconnected via a communications network 53 or other communications link. Each individual computer or machine is provided with a corresponding modifier 51. Each individual computer is also provided with a communications port which connects to the communications network. The communications network 53 or path can be any electronic signalling, data, or digital communications network or path and is preferably slow speed, and thus low cost, communications path, such as a network connection over the Internet or any common networking configurations including communication ports known or available as of the date of this application such as ETHERNET or INFINIBAND and extensions and improvements, thereto.

As a consequence of the above described arrangement, if each of the machines M1, M2, ..., Mn has say an internal or local memory capability of 10MB, then the total memory available to the application code 50 in its entirety is not, as one might expect, the number of machines (n) times 10MB. Nor is it the additive combination of the internal memory capability of all n machines. Instead it is either 10MB, or some number greater than 10MB but less than n x 10MB. In the situation where the internal memory capacities of the machines are different, which is permissible, then in the case where the internal memory in one machine is smaller than the internal memory capability of at least one other of the machines, then the size of the smallest memory of any of the machines may be used as the maximum memory capacity of the machines when such memory (or a portion thereof) is to be treated as 'common' memory (i.e. similar equivalent memory on each of the machines M1...Mn) or otherwise used to execute the common application code.

However, even though the manner that the internal memory of each machine is treated may initially appear to be a possible constraint on performance, how this results in improved operation and performance will become apparent hereafter. Naturally, each machine M1, M2...Mn has a private (i.e. 'non-common') internal memory capability. The private internal memory capability of the machines M1, M2, ..., Mn are normally approximately equal but need not be. It may also be advantageous to select the amounts of internal memory in each machine to achieve a desired performance level in each machine and across a constellation or network of connected or coupled plurality of machines, computers, or information appliances M1, M2, ..., Mn. Having described these internal and common memory considerations, it will be apparent in light of the description provided herein that the amount of memory that can be common between machines is not a limitation.

In some embodiments, some or all of the plurality of individual computers or machines can be contained within a single housing or chassis (such as so-called "blade servers" manufactured by Hewlett-Packard Development Company, Intel Corporation, IBM Corporation and others) or implemented on a single printed circuit board or even within a single chip or chip set.

When implemented in a non-JAVA language or application code environment, the generalized platform, and/or virtual machine and/or machine and/or runtime system is able to operate application code 50 in the language(s) (possibly including for example, but not limited to any one or more of source-code languages, intermediate-code languages, object-code languages, machine-code languages, and any other code languages) of that platform and/or virtual machine and/or machine and/or runtime system environment, and utilize the platform, and/or virtual machine and/or machine and/or runtime system and/or language architecture irrespective of the machine manufacturer and the internal details of the machine. It will also be appreciated that the platform and/or runtime system can include virtual machine and non-virtual machine software and/or firmware architectures, as well as hardware and direct hardware coded applications and implementations.

For a more general set of virtual machine or abstract machine environments, and for current and future computers and/or computing machines and/or information appliances or processing systems, and that may not utilize or require utilization of either classes and/or objects, the inventive structure, method and computer program and computer program product are still applicable. Examples of computers and/or computing machines that do not utilize either classes and/or objects include for example, the x86 computer architecture manufactured by Intel Corporation and others, the SPARC computer architecture manufactured by Sun Microsystems, Inc and others, the Power PC computer architecture manufactured by International Business Machines Corporation and others, and the personal computer products made by Apple Computer, Inc., and others. For these types of computers, computing machines, information appliances, and the virtual machine or virtual computing environments implemented thereon that do not utilize the idea of classes or objects, may be generalized for example to include primitive data types (such as integer data types, floating point data types, long data types, double data types, string data types, character data types and Boolean data types), structured data types (such a arrays and records) derived types, or other code or data structures of procedural languages or other languages and environments such as functions, pointers, components, modules, structures, reference and unions. These structures and procedures when applied in combination when required, maintain a computing environment where memory locations, address ranges, objects, classes, assets, resources, or any other procedural or structural aspect of a computer or computing environment are where required created, maintained, operated, and deactivated or deleted in a coordinated, coherent, and consistent manner across the plurality of individual machines M1, M2...Mn.

This analysis or scrutiny of the application code 50 can take place either prior to loading the application program code 50, or during the application program code 50 loading procedure, or even after the application program code 50 loading procedure. It may be likened to an instrumentation, program transformation, translation, or compilation procedure in that the application code can be instrumented with additional instructions, and/or otherwise modified by meaning-preserving program manipulations, and/or optionally translated from an input code language to a different code language (such as for example from source-code language or intermediate-code language to object-code language or machine-code language). In this connection it is understood that the term compilation normally or conventionally involves a change in code or language, for example, from source code to object code or from one language to another language. However, in the present instance the term "compilation" (and its grammatical equivalents) is not so restricted and can also include or embrace modifications within the same code or language. For example, the compilation and its equivalents are understood to encompass both ordinary compilation (such as for example by way of illustration but not limitation, from source-code to object code), and compilation from source-code to source-code, as well as compilation from object-code to object code, and any altered combinations therein. It is also inclusive of so-called "intermediary-code languages" which are a form of "pseudo object-code".

By way of illustration and not limitation, in one embodiment, the analysis or scrutiny of the application code 50 takes place during the loading of the application program code such as by the operating system reading the application code 50 from the hard disk or other storage device or source and copying it into memory and preparing to begin execution of the application program code. In another embodiment, in a JAVA virtual machine, the analysis or scrutiny may take place during the class loading procedure of the java.lang.ClassLoader.loadClass method (e.g. "java.lang.ClassLoader.loadClass()").

Alternatively, the analysis or scrutiny of the application code 50 may take place even after the application program code loading procedure, such as after the operating system has loaded the application code into memory, or optionally even after execution of the relevant corresponding portion of the application program code has started, such as for example after the JAVA virtual machine has loaded the application code into the virtual machine via the "java.lang.ClassLoader.loadClass()" method and optionally commenced execution.

Persons skilled in the computing arts will be aware of various possible techniques that may be used in the modification of computer code, including but not limited to instrumentation, program transformation, translation, or compilation means.

One such technique is to make the modification(s) to the application code, without a preceding or consequential change of the language of the application code. Another such technique is to convert the original code (for example, JAVA language source-code) into an intermediate representation (or intermediate-code language, or pseudo code), such as JAVA byte code. Once this conversion takes place the modification is made to the byte code and then the conversion may be reversed. This gives the desired result of modified JAVA code.

A further possible technique is to convert the application program to machine code, either directly from source-code or via the abovementioned intermediate language or through some other intermediate means. Then the machine code is modified before being loaded and executed. A still further such technique is to convert the original code to an intermediate representation, which is thus modified and subsequently converted into machine code.

The present invention encompasses all such modification routes and also a combination of two, three or even more, of such routes.

The DRT or other code modifying means is responsible for creating or replication a memory structure and contents on each of the individual machines M1, M2...Mn that permits the plurality of machines to interoperate. In some embodiments this replicated memory structure will be identical. Whilst in other embodiments this memory structure will have portions that are identical and other portions that are not. In still other embodiments the memory structures are different only in format or storage conventions such as Big Endian or Little Endian formats or conventions.

These structures and procedures when applied in combination when required, maintain a computing environment where the memory locations, address ranges, objects, classes, assets, resources, or any other procedural or structural aspect of a computer or computing environment are where required created, maintained, operated, and deactivated or deleted in a coordinated, coherent, and consistent manner across the plurality of individual machines M1, M2...Mn.

Therefore the terminology "one", "single", and "common" application code or program includes the situation where all machines M1, M2...Mn are operating or executing the same program or code and not different (and unrelated) programs, in other words copies or replicas of same or substantially the same application code are loaded onto each of the interoperating and connected machines or computers.

In conventional arrangements utilising distributed software, memory access from one machine's software to memory physically located on another machine takes place via the network interconnecting the machines. However, because the read and/or write memory access to memory physically located on another computer require the use of the slow network interconnecting the computers, in these configurations such memory accesses can result in substantial delays in memory read/write processing operations, potentially of the order of $10^6 - 10^7$ cycles of the central processing unit of the machine. Ultimately this delay is dependent upon numerous factors, such as for example, the speed, bandwidth, and/or latency of the communication network. This in large part accounts for the diminished performance of the multiple interconnected machines in the prior art arrangement.

However, in the present arrangement all reading of memory locations or data is satisfied locally because a current value of all (or some subset of all) memory locations is stored on the machine carrying out the processing which generates the demand to read memory.

Similarly, all writing of memory locations or data is satisfied locally because a current value of all (or some subset of all) memory locations is stored on the machine carrying out the processing which generates the demand to write to memory.

Such local memory read and write processing operation can typically be satisfied with $10^2 - 10^3$ cycles of the central processing unit. Thus, in practice there is substantially less waiting for memory accesses which involves and/or writes.

The invention is transport, network, and communications path independent, and does not depend on how the communication between machines or DRTs takes place. In one embodiment, even electronic mail (email) exchanges between machines or DRTs may suffice for the communications.

Turning now to Fig. 9, during the loading procedure 75, the program 50 being loaded to create each JAVA virtual machine M1, M2,...Mn is modified. This modification commences at 90 in Fig. 9 and involves the initial step 91 of detecting all memory locations (termed fields in JAVA - but equivalent terms are used in other languages) in the application 50 being loaded. Such memory locations need to be identified for subsequent processing at steps 92 and 93. The DRT 71/1,...DRT71/n during the loading procedure 75 creates a list of all the memory locations thus identified, the JAVA fields being listed by object and class. Both volatile and synchronous fields are listed.

The next phase (designated 92 in Fig. 9) of the modification procedure is to search through the executable application code in order to locate every processing activity that manipulates or changes field values corresponding to the list generated at step 91 and thus writes to fields so the value at the corresponding memory location is changed. When such an operation (typically putstatic or putfield in the JAVA language) is detected which changes the field value, then an "updating propagation routine" is inserted by step 93 at this place in the program to ensure that all other machines are notified that the value of the field has changed. Thereafter, the loading procedure continues in a normal way as indicated by step 94 in Fig. 9.

An alternative form of initial modification during loading is illustrated in Fig. 10. Here the start and listing steps 90 and 91 and the searching step 92 are the same as in Fig. 9. However, rather than insert the "updating propagation routine" as in step 93 in which the processing thread carries out the updating, instead an "alert routine" is inserted at step 103. The "alert routine" instructs a thread or threads not used in processing and allocated to the DRT, to carry out the necessary propagation. This step 103 is a quicker alternative which results in lower overhead.

Once this initial modification during the loading procedure has taken place, then either one of the multiple thread processing operations illustrated in Figs. 11 and 12 takes place. As seen in Fig. 11, multiple thread processing 110 on the machines consisting of threads 111/1...111/4 is occurring and the processing of the second thread 111/2 (in this example) results in that thread 111/2 becoming aware at step 113 of a change of field value. At this stage the normal processing of that thread 111/2 is halted at step 114, and the same thread 111/2 notifies all other machines M2...Mn via the network 53 of the identity of the changed field and the changed value which occurred at step 113. At the end of that communication procedure, the thread 111/2 then resumes the processing at step 115 until the next instance where there is a change of field value.

In the alternative arrangement illustrated in Fig. 12, once a thread 121/2 has become aware of a change of field value at step 113, it instructs DRT processing 120 (as indicated by step 125 and arrow 127) that another thread(s) 121/1 allocated to the DRT processing 120 is to propagate in accordance with step 128 via the network 53 to all other machines M2...Mn the identity of the changed field and the changed value detected at step 113. This is an operation which can be carried out quickly and thus the processing of the initial thread 111/2 is only interrupted momentarily as indicated in step 125 before the thread 111/2 resumes processing in step 115. The other thread 121/1 which has been notified of the change (as indicated by arrow 127) then communicates that change as indicated in step 128 via the network 53 to each of the other machines M2...Mn.

This second arrangement of Fig. 12 makes better utilisation of the processing power of the various threads 111/1...111/3 and 121/1 (which are not, in general, subject to equal demands) and gives better scaling with increasing size of "n", (n being an integer greater than or equal to 2 which represents the total number of machines which are connected to the network 53 and which run the application program 50 simultaneously). Irrespective of which arrangement is used, the changed field and identities and values detected at step 113 are propagated to all the other machines M2...Mn on the network.

This is illustrated in Fig. 13 where the DRT 71/1 and its thread 121/1 of Fig. 12 (represented by step 128 in Fig. 13) sends via the network 53 the identity and changed value of the listed memory location generated at step 113 of Fig. 12 by processing in machine M1, to each of the other machines M2...Mn.

Each of the other machines M2...Mn carries out the action indicated by steps 135 and 136 in Fig. 13 for machine Mn by receiving the identity and value pair from the network 53 and writing the new value into the local corresponding memory location.

In the prior art arrangement in Fig. 3 utilising distributed software, memory accesses from one machine's software to memory physically located on another machine are permitted by the network interconnecting the machines. However, such memory accesses can result in delays in processing of the order of $10^6 - 10^7$ cycles of the central processing unit of the machine. This in large part accounts for the diminished performance of the multiple interconnected machines.

However, in the present arrangement as described above in connection with Fig. 8, it will be appreciated that all reading of data is satisfied locally because the current value of all fields is stored on the machine carrying out the processing which generates the demand to read memory. Such local processing can be satisfied within $10^2 - 10^3$ cycles of the central processing unit. Thus, in practice, there is substantially no waiting for memory accesses which involves reads.

However, most application software reads memory frequently but writes to memory relatively infrequently. As a consequence, the rate at which memory is being written or re-written is relatively slow compared to the rate at which memory is being read. Because of this slow demand for writing or re-writing of memory, the fields can be continually updated at a relatively low speed via the inexpensive commodity network 53, yet this low speed is sufficient to meet the application program's demand for writing to memory. The result is that the performance of the Fig. 8 arrangement is vastly superior to that of Fig. 3.

In a further modification in relation to the above, the identities and values of changed fields can be grouped into batches so as to further reduce the demands on the communication speed of the network 53 interconnecting the various machines.

It will also be apparent to those skilled in the art that in a table created by each DRT 71 when initially recording the fields, for each field there is a name or identity which is common throughout the network and which the network recognises. However, in the individual machines the memory location corresponding to a given named field will vary over time since each machine will progressively store changed field values at different locations according to its own internal processes. Thus the table in each of the DRTs will have, in general, different memory locations but each global "field name" will have the same "field value" stored in the different memory locations.

It will also be apparent to those skilled in the art that the abovementioned modification of the application program during loading can be accomplished in up to five ways by:

(i) re-compilation at loading,
(ii) by a pre-compilation procedure prior to loading,
(iii) compilation prior to loading,
(iv) a "just-in-time" compilation, or
(v) re-compilation after loading (but, or for example, before execution of the relevant or corresponding application code in a distributed environment).

Traditionally the term "compilation" implies a change in code or language, eg from source to object code or one language to another. Clearly the use of the term "compilation" (and its grammatical equivalents) in the present specification is not so restricted and can also include or embrace modifications within the same code or language.

In the first embodiment, a particular machine, say machine M2, loads the application code on itself, modifies it, and then loads each of the other machines M1, M3 ... Mn (either sequentially or simultaneously) with the modified code. In this arrangement, which may be termed "master/slave", each of machines M1, M3, ... Mn loads what it is given by machine M2.

In a still further embodiment, each machine receives the application code, but modifies it and loads the modified code on that machine. This enables the modification carried out by each machine to be slightly different being optimized based upon its architecture and operating system, yet still coherent with all other similar modifications.

In a further arrangement, a particular machine, say M1, loads the unmodified code and all other machines M2, M3 ... Mn do a modification to delete the original application code and load the modified version.

In all instances, the supply can be branched (ie M2 supplies each of M1, M3, M4, etc directly) or cascaded or sequential (ie M2 applies M1 which then supplies M3 which then supplies M4, and so on).

In a still further arrangement, the machines M1 to Mn, can send all load requests to an additional machine (not illustrated) which is not running the application program, which performs the modification via any of the aforementioned methods, and returns the modified routine to each of the machines M1 to Mn which then load the modified routine locally. In this arrangement, machines M1 to Mn forward all load requests to this additional machine which returns a modified routine to each machine. The modifications performed by this additional machine can include any of the modifications covered under the scope of the present invention.

Persons skilled in the computing arts will be aware of at least four techniques used in creating modifications in computer code. The first is to make the modification in the original (source) language. The second is to convert the original code (in say JAVA) into an intermediate representation (or intermediate language). Once this conversion takes place the modification is made and then the conversion is reversed. This gives the desired result of modified JAVA code.

The third possibility is to convert to machine code (either directly or via the abovementioned intermediate language). Then the machine code is modified before being loaded and executed. The fourth possibility is to convert the original code to an intermediate representation, which is then modified and subsequently converted into machine code.

The present invention encompasses all four modification routes and also a combination of two, three or even all four, of such routes.

MEMORY MANAGEMENT AND REPLICATION

In connection with FIG. 5, in accordance with a preferred embodiment of the present invention a single application code 50 (sometimes more informally referred to as the application or the application program) can be operated simultaneously on a number of machines M1, M2...Mn interconnected via a communications network or other communications link or path 53. By way of example but not limitation, one application code or program 50 would be a single common application program on the machines, such as Microsoft Word, as opposed to different applications on each machine, such as Microsoft Word on machine M1, and Microsoft PowerPoint on machine M2, and Netscape Navigator on machine M3 and so forth. Therefore the terminology "one", "single", and "common" application code or program is used to try and capture this situation where all machines M1, ..., Mn are operating or executing the same program or code and not different (and unrelated) programs. In other words copies or replicas of same or substantially the same application code is loaded onto each of the interoperating and connected machines or computers. As the characteristics of each machine or computer may differ, the application code 50 may be modified before loading, during the loading process, or after the loading process to provide a customization or modification of the code on each machine. Some dissimilarity between the programs may be permitted so long as the other requirements for interoperability, consistency, and coherency as described herein can be maintain. As it will become apparent hereafter, each of the machines M1, M2...Mn operates with the same application code 50 on each machine M1, M2...Mn and thus all of the machines M1, M2, ..., Mn have the same or substantially the same application code 50 usually with a modification that may be machine specific.

Similarly, each of the machines M1, M2, ..., Mn operates with the same (or substantially the same or similar) modifier 51 on each machine M1, M2, ...,Mn and thus all of the machines M1, M2...Mn have the same (or substantially the same or similar) modifier 51 with the modifier of machine M1 being designated 51/1 and the modifier of machine M2 being designated 51/2, etc. In addition, before or during the loading of, or preceding the execution of, or even after execution has commenced, the application code 50 on each machine M1, M2...Mn is modified by the corresponding modifier 51 according to the same rules (or substantially the same rules since minor optimizing changes are permitted within each modifier 51/1, 51/2, ..., 51/n).

As will become more apparent in light of the further description provided herein, one of the features of the invention is to make it appear that one application program instance of application code 50 is executing simultaneously across all of the plurality of machines M1, M2, ..., Mn. As will be described in considerable detail hereinafter, the instant invention achieves this by running the same application program code (for example, Microsoft Word or Adobe Photoshop CS2) on each machine, but modifying the executable code of that application program on each machine such that each executing occurrence (or 'local instance') on each one of the machines M1...Mn coordinates its local operations with the operations of the respective occurrences on each one of the other machines such that each occurrence on each one of the plurality of machines function together in a consistent, coherent and coordinated manner so as to give the appearance of being one global instance (or occurrence) of the application program and program code(i.e., a "meta-application").

As a consequence of the above described arrangement, if each of the machines M1, M2, ..., Mn has, say, an internal memory capability of 10MB, then the total memory available to each application code 50 is not necessarily, as one might expect the number of machines (n) times 10MB, or alternatively the additive combination of the internal memory capability of all n machines, but rather or still may only be 10MB. In the situation where the internal memory capacities of the machines are different, which is permissible, then in the case where the internal memory in one machine is smaller than the internal memory capability of at least one other of the machines, then the size of the smallest memory of any of the machines may be used as the maximum memory capacity of the machines when such memory (or a portion thereof) is to be treated as a 'common' memory (i.e. similar equivalent memory on each of the machines M1...Mn) or otherwise used to execute the common application code.

However, even though the manner that the internal memory of each machine is treated may initially appear to be a possible constraint on performance, how this results in improved operation and performance will become apparent hereafter. Naturally, each machine M1, M2...Mn has a private (i.e. 'non-common') internal memory capability. The private internal memory capability of the machines M1, M2, ..., Mn are normally approximately equal but need not be. It may also be advantageous to select the amounts of internal memory in each machine to achieve a desired performance level in each machine and across a constellation or network of connected or coupled plurality of machines, computers, or information appliances M1, M2, ..., Mn. Having described these internal and common memory considerations, it will be apparent in light of the description provided herein that the amount of memory that can be common between machines is not a limitation of the invention.

It is known from the prior art to operate a single computer or machine (produced by one of various manufacturers and having an operating system operating in one of various different languages) in a particular language of the application, by creating a virtual machine as schematically illustrated in FIG. 6. The code and data and virtual machine configuration or arrangement of FIG. 6 takes the form of the application code 50 written in the Java language and executing within a Java Virtual Machine 61. Thus, where the intended language of the application is the language JAVA, a JAVA virtual machine is used which is able to operate code in JAVA irrespective of the machine manufacturer and internal details of the machine. For further details see "The JAVA Virtual Machine Specification" 2nd Edition by T. Lindholm & F. Yellin of Sun Microsystems Inc. of the USA, which is incorporated by reference herein.

This conventional art arrangement of FIG. 6 is modified in accordance with embodiments of the present invention by the provision of an additional facility which is conveniently termed "distributed run time" or "distributed run time system" DRT 71 and as seen in FIG. 7.

In FIG. 7, the application code 50 is loaded onto the Java Virtual Machine 72 in cooperation with the distributed runtime system 71, through the loading procedure indicated by arrow 75. As used herein the terms distributed runtime and the distributed run time system are essentially synonymous, and by means of illustration but not limitation are generally understood to include library code and processes which support software written in a particular language running on a particular platform. Additionally, a distributed runtime system may also include library code and processes which support software written in a particular language running within a particular distributed computing environment. The runtime system typically deals with the details of the interface between the program and the operation system such as system calls, program start-up and termination, and memory management. For purposes of background, a conventional Distributed Computing Environment (DCE) that does not provide the capabilities of the inventive distributed run time or distributed run time system 71 required in the invention is available from the Open Software Foundation. This Distributed Computing Environment (DCE) performs a form of computer-to-computer communication for software running on the machines, but among its many limitations, it is not able to implement the modification or communication operations of this invention. Among its functions and operations, the inventive DRT 71 coordinates the particular communications between the plurality of machines M1, M2, ..., Mn. Moreover, the inventive distributed runtime 71 comes into operation during the loading procedure indicated by arrow 75 of the JAVA application 50 on each JAVA virtual machine 72 of machines JVM#1, JVM#2,...JVM#n. The sequence of operations during loading will be described hereafter in relation to FIG. 9. It will be appreciated in light of the description provided herein that although many examples and descriptions are provided relative to the JAVA language and JAVA virtual machines so that the reader may get the benefit of specific examples, the invention is not restricted to either the JAVA language or JAVA virtual machines, or to any other language, virtual machine, machine, or operating environment.

FIG. 8 shows in modified form the arrangement of FIG. 5 utilising JAVA virtual machines, each as illustrated in FIG. 7. It will be apparent that again the same application code 50 is loaded onto each machine M1, M2...Mn. However, the communications between each machine M1, M2, ..., Mn, and indicated by arrows 83, although physically routed through the machine hardware, are advantageously controlled by the individual DRT's 71/1...71/n within each machine. Thus, in practice this may be conceptionalised as the DRT's 71/1, ..., 71/n communicating with each other via the network or other communications link 73 rather than the machines M1, M2, ..., Mn communicating directly with themselves or each other. Actually, the invention contemplates and included either this direct communication between machines M1, M2, ..., Mn or DRTs 71/1, 71/2, ..., 71/n or a combination of such communications. The inventive DRT 71 provides communication that is transport, protocol, and link independent.

It will be appreciated in light of the description provided herein that there are alternative implementations of the modifier 51 and the distributed run time 71. For example, the modifier 51 may be implemented as a component of or within the distributed run time 71, and therefore the DRT 71 may implement the functions and operations of the modifier 51. Alternatively, the function and operation of the modifier 51 may be implemented outside of the structure, software, firmware, or other means used to implement the DRT 71. In one embodiment, the modifier 51 and DRT 71 are implemented or written in a single piece of computer program code that provides the functions of the DRT and modifier. The modifier function and structure therefore maybe subsumed into the DRT and considered to be an optional component. Independent of how implemented, the modifier function and structure is responsible for modifying the executable code of the application code program, and the distributed run time function and structure is responsible for implementing communications between and among the computers or machines. The communications functionality in one embodiment is implemented via an intermediary protocol layer within the computer program code of the DRT on each machine. The DRT may for example implement a communications stack in the JAVA language and use the Transmission Control Protocol/Internet Protocol (TCP/IP) to provide for communications or talking between the machines. Exactly how these functions or operations are implemented or divided between structural and/or procedural elements, or between computer program code or data structures within the invention are less important than that they are provided.

However, in the arrangement illustrated in FIG. 8, (and also in FIGS. 31-32), a plurality of individual computers or machines M1, M2, ..., Mn are provided, each of which are interconnected via a communications network 53 or other communications link and each of which individual computers or machines provided with a modifier 51 (See in

FIG. 5) and realised by or in for example the distributed run time (DRT) 71 (See FIG. 8) and loaded with a common application code 50. The term common application program is to be understood to mean an application program or application program code written to operate on a single machine, and loaded and/or executed in whole or in part on each one of the plurality of computers or machines M1, M2...Mn, or optionally on each one of some subset of the plurality of computers or machines M1,M2...Mn. Put somewhat differently, there is a common application program represented in application code 50, and this single copy or perhaps a plurality of identical copies are modified to generate a modified copy or version of the application program or program code, each copy or instance prepared for execution on the plurality of machines. At the point after they are modified they are common in the sense that they perform similar operations and operate consistently and coherently with each other. It will be appreciated that a plurality of computers, machines, information appliances, or the like implementing the features of the invention may optionally be connected to or coupled with other computers, machines, information appliances, or the like that do not implement the features of the invention.

Essentially in at least one embodiment the modifier 51 or DRT 71 or other code modifying means is responsible for modifying the application code 50 so that it may execute memory manipulation operations, such as memory putstatic and putfield instructions in the JAVA language and virtual machine environment, in a coordinated, consistent, and coherent manner across and between the plurality of individual machines M1...Mn. It follows therefore that in such a computing environment it is necessary to ensure that each of memory location is manipulated in a consistent fashion (with respect to the others).

In some embodiments, some or all of the plurality of individual computers or machines may be contained within a single housing or chassis (such as so-called "blade servers" manufactured by Hewlett-Packard Development Company, Intel Corporation, IBM Corporation and others) or implemented on a single printed circuit board or even within a single chip or chip set.

A machine (produced by any one of various manufacturers and having an operating system operating in any one of various different languages) can operate in the particular language of the application program code 50, in this instance the JAVA language. That is, a JAVA virtual machine 72 is able to operate application code 50 in the JAVA language, and utilize the JAVA architecture irrespective of the machine manufacturer and the internal details of the machine.

When implemented in a non-JAVA language or application code environment, the generalized platform, and/or virtual machine and/or machine and/or runtime system is able to operate application code 50 in the language(s) (possibly including for example, but not limited to any one or more of source-code languages, intermediate-code languages, object-code languages, machine-code languages, and any other code languages) of that platform, and/or virtual machine and/or machine and/or runtime system environment, and utilize the platform, and/or virtual machine and/or machine and/or runtime system and/or language architecture irrespective of the machine manufacturer and the internal details of the machine. It will also be appreciated in light of the description provided herein that platform and/or runtime system may include virtual machine and non-virtual machine software and/or firmware architectures, as well as hardware and direct hardware coded applications and implementations.

For a more general set of virtual machine or abstract machine environments, and for current and future computers and/or computing machines and/or information appliances or processing systems, and that may not utilize or require utilization of either classes and/or objects, the inventive structure, method, and computer program and computer program product are still applicable. Examples of computers and/or computing machines that do not utilize either classes and/or objects include for example, the x86 computer architecture manufactured by Intel Corporation and others, the SPARC computer architecture manufactured by Sun Microsystems, Inc and others, the PowerPC computer architecture manufactured by International Business Machines Corporation and others, and the personal computer products made by Apple Computer, Inc., and others. For these types of computers, computing machines, information appliances, and the virtual machine or virtual computing environments implemented thereon that do not utilize the idea of classes or objects, may be generalized for example to include primitive data types (such as integer data types, floating point data types, long data types, double data types, string data types, character data types and Boolean data types), structured data types (such as arrays and records) derived types, or other code or data structures of procedural languages or other languages and environments such as functions, pointers, components, modules, structures, references and unions.

Turning now to FIGS. 7 and 9, during the loading procedure 75, the application code 50 being loaded onto or into each JAVA virtual machine 72 is modified by DRT 71. This modification commences at Step 90 in FIG. 9 and involves the initial step 91 of preferably scrutinizing or analysing the code and detecting all memory locations addressable by the application code 50, or optionally some subset of all memory locations addressable by the application code 50; such as for example named and unnamed memory locations, variables (such as local variables, global variables, and formal arguments to subroutines or functions), fields, registers, or any other address space or range of addresses which application code 50 may access. Such memory locations in some instances need to be identified for subsequent processing at steps 92 and 93. In some embodiments, where a list of detected memory locations is required for further processing, the DRT 71 during the loading procedure 75 creates a list of all the memory locations thus identified. In one embodiment, the memory locations in the form of JAVA fields are listed by object and class, however, the memory locations, fields, or the like may be listed or organized in any manner so long as they comport with the architectural and programming requirements of the system on which the program is to be used and the principles of the invention described herein. This detection is optional and not required in all embodiments of the invention. It may be noted that the DRT is at least in part fulfilling the roll of the modifier 51.

The next phase (designated Step 92 in FIG. 9) [Step 92] of the modification procedure is to search through the application code 50 in order to locate processing activity or activities that manipulate or change values or contents of any listed memory location (for example, but not limited to JAVA fields) corresponding to the list generated at step 91 when required. Preferably, all processing activities that manipulate or change any one or more values or contents of any one or more listed memory locations, are located.

When such a processing activity or operation (typically "putstatic" or "putfield" in the JAVA language, or for example, a memory assignment operation, or a memory write operation, or a memory manipulation operation, or more generally operations that otherwise manipulate or change value(s) or content(s) of memory or other addressable areas), is detected which changes the value or content of a listed or detected memory location , then an "updating propagation routine" is inserted by step 93 in the application code 50 corresponding to the detected memory manipulation operation, to communicate with all other machines in order to notify all other machines of the identity of the manipulated memory location, and the updated, manipulated or changed value(s) or content(s) of the manipulated memory location. The inserted "updating propagation routine" preferably takes the form of a method, function, procedure, or similar subroutine call or operation to a network communications library of DRT 71. Alternatively, the "updating propagation routine" may take the optional form of a code-block (or other inline code form) inserted into the application code instruction stream at, after, before, or otherwise corresponding to the detected manipulation instruction or operation. And preferably, in a multi-tasking or parallel processing machine environment (and in some embodiments inclusive or exclusive of operating system), such as a machine environment capable of potentially simultaneous or concurrent execution of multiple or different threads or processes, the "updating propagation routine" may execute on the same thread or process or processor as the detected memory manipulation operation of step 92. Thereafter, the loading procedure continues, by loading the modified application code 50 on the machine 72 in place of the unmodified application code 50, as indicated by step 94 in FIG. 9.

An alternative form of modification during loading is illustrated in the illustration of FIG. 10. Here the start and listing steps 90 and 91 and the searching step 92 are the same as in FIG. 9. However, rather than insert the "updating propagation routine" into the application code 50 corresponding to the detected memory manipulation operation identified in step 92, as is indicated in step 93, in which the application code 50, or network communications library code 71 of the DRT executing on the same thread or process or processor as the detected memory manipulation operation, carries out the updating, instead an "alert routine" is inserted corresponding to the detected memory manipulation operation, at step 103. The "alert routine" instructs, notifies or otherwise requests a different and potentially simultaneously or concurrently executing thread or process or processor not used to perform the memory manipulation operation (that is, a different thread or process or processor than the thread or process or processor which manipulated the memory location), such as a different thread or process allocated to the DRT 71, to carry out the notification , propagation, or communication of all other machines of the identity of the manipulated memory location, and the updated, manipulated or changed value(s) or content(s) of the manipulated memory location.

Once this modification during the loading procedure has taken place and execution begins of the modified application code 50, then either the steps of FIG. 11 or FIG. 12 take place. FIG. 11 (and the steps 112, 113, 114, and 115 therein) correspond to the execution and operation of the modified application code 50 when modified in accordance with the procedures set forth in and described relative to FIG. 9. FIG. 12 on the other hand (and the steps 112, 113, 125, 127, and 115 therein) set forth therein correspond to the execution and operation of the modified application code 50 when modified in accordance with FIG. 10.

This analysis or scrutiny of the application code 50 can take place either prior to loading the application program code 50, or during the application program code 50 loading procedure, or even after the application program code 50 loading procedure. It may be likened to an instrumentation, program transformation, translation, or compilation procedure in that the application code may be instrumented with additional instructions, and/or otherwise modified by meaning-preserving program manipulations, and/or optionally translated from an input code language to a different code language (such as for example from source-code language or intermediate-code language to object-code language or machine-code language), and with the understanding that the term compilation normally or conventionally involves a change in code or language, for example, from source code to object code or from one language to another language. However, in the present instance the term "compilation" (and its grammatical equivalents) is not so restricted and can also include or embrace modifications within the same code or language. For example, the compilation and its equivalents are understood to encompass both ordinary compilation (such as for example by way of illustration but not limitation, from source-code to object-code), and compilation from source-code to source-code, as well as compilation from object-code to object-code, and any altered combinations therein. It is also inclusive of so-called "intermediary-code languages" which are a form of "pseudo object-code".

By way of illustration and not limitation, in one embodiment, the analysis or scrutiny of the application code 50 may take place during the loading of the application program code such as by the operating system reading the application code from the hard disk or other storage device or source and copying it into memory and preparing to begin execution of the application program code. In another embodiment, in a JAVA virtual machine, the analysis or scrutiny may take place during the class loading procedure of the java.lang.ClassLoader loadClass method (e.g., "java.lang.ClassLoader.loadClass()").

Alternatively, the analysis or scrutiny of the application code 50 may take place even after the application program code loading procedure, such as after the operating system has loaded the application code into memory, or optionally even after execution of the relevant corresponding portion of the application program code has started, such as for example after the JAVA virtual machine has loaded the application code into the virtual machine via the "java.lang.ClassLoader.loadClass()" method and optionally commenced execution.

As seen in FIG. 11, a multiple thread processing machine environment 110, on each one of the machines M1, ..., Mn and consisting of threads 111/1...111/4 exists. The processing and execution of the second thread 111/2 (in this example) results in that thread 111/2 manipulating a memory location at step 113, by writing to a listed memory location. In accordance with the modifications made to the application code 50 in the steps 90-94 of FIG. 9, the application code 50 is modified at a point corresponding to the write to the memory location of step 113, so that it propagates, notifies, or communicates the identity and changed value of the manipulated memory location of step 113 to the other machines M2, ..., Mn via network 53 or other communication link or path, as indicated at step 114. At this stage the processing of the application code 50 of that thread 111/2 is or may be altered and in some instances interrupted at step 114 by the executing of the inserted "updating propagation routine", and the same thread 111/2 notifies, or propagates, or communicates to all other machines M2, ..., Mn via the network 53 or other communications link or path of the identity and changed value of the manipulated memory location of step 113. At the end of that notification, or propagation, or communication procedure 114, the thread 111/2 then resumes or continues the processing or the execution of the modified application code 50 at step 115.

In the alternative arrangement illustrated in FIG. 12, a multiple thread processing machine environment 110 comprising or consisting of threads 111/1, ..., 111/3, and a simultaneously or concurrently executing DRT processing environment 120 consisting of the thread 121/1 as illustrated, or optionally a plurality of threads, is executing on each one of the machines M1,...Mn. The processing and execution of the modified application code 50 on thread 111/2 results in a memory manipulation operation of step 113, which in this instance is a write to a listed memory location. In accordance with the modifications made to the application code 50 in the steps 90, 91, 92, 103, and 94 of FIG. 9, the application code 50 is modified at a point corresponding to the write to the memory location of step 113, so that it requests or otherwise notifies the threads of the DRT processing environment 120 to notify, or propagate, or communicate to the other machines M2, ..., Mn of the identity and changed value of the manipulated memory location of step 113, as indicated at steps 125 and 128 and arrow 127. In accordance with this modification, the thread 111/2 processing and executing the modified application code 50 requests a different and potentially simultaneously or concurrently executing thread or process (such as thread 121/1) of the DRT processing environment 120 to notify the machines M2, ..., Mn via network 53 or other communications link or path of the identity and changed value of the manipulated memory location of step 113, as indicated in step 125 and arrow 127. In response to this request of step 125 and arrow 127, a different and potentially simultaneously or concurrently executing thread or process 121/1 of the DRT processing environment 120 notifies the machines M2, ..., Mn via network 53 or other communications link or path of the identity and changed value of the manipulated memory location of step 113, as requested of it by the modified application code 50 executing on thread 111/2 of step 125 and arrow 127.

When compared to the earlier described step 114 of thread 111/2 of FIG. 11, step 125 of thread 111/2 of FIG. 12 can be carried out quickly, because step 114 of thread 111/2 must notify and communicate with machines M2, ..., Mn via the relatively slow network 53 (relatively slow for example when compared to the internal memory bus 4 of FIG. 1 or the global memory 13 of FIG. 2) of the identity and changed value of the manipulated memory location of step 113, whereas step 125 of thread 111/2 does not communicate with machines M2, ..., Mn via the relatively slow network 53. Instead, step 125 of thread 111/2 requests or otherwise notifies a different and potentially simultaneously or concurrently executing thread 121/1 of the DRT processing environment 120 to perform the notification and communication with machines M2, ..., Mn via the relatively slow network 53 of the identify and changed value of the manipulated memory location of step 113, as indicated by arrow 127. Thus thread 111/2 carrying out step 125 is only interrupted momentarily before the thread 111/2 resumes or continues processing or execution of modified application code in step 115. The other thread 121/1 of the DRT processing environment 120 then communicates the identity and changed value of the manipulated memory location of step 113 to machines M2, ..., Mn via the relatively slow network 53 or other relatively slow communications link or path.

This second arrangement of FIG. 12 makes better utilisation of the processing power of the various threads 111/1...111/3 and 121/1 (which are not, in general, subject to equal demands). Irrespective of which arrangement is used, the identity and change value of the manipulated memory location(s) of step 113 is (are) propagated to all the other machines M2...Mn on the network 53 or other communications link or path.

This is illustrated in FIG. 13 where step 114 of Fig 11, or the DRT 71/1 (corresponding to the DRT processing environment 120 of Fig 12) and its thread 121/1 of FIG. 12 (represented by step 128 in FIG. 13), send, via the network 53 or other communications link or path, the identity and changed value of the manipulated memory location of step 113 of FIGS. 11 and 12, to each of the other machines M2, ..., Mn.

With reference to FIG. 13, each of the other machines M2, ..., Mn carries out the action of receiving from the network 53 the identity and changed value of, for example, the manipulated memory location of step 113 from machine M1, indicated by step 135, and writes the value received at step 135 to the local memory location corresponding to the identified memory location received at step 135, indicated by step 136.

In the conventional arrangement in FIG. 3 utilising distributed software, memory access from one machine's software to memory physically located on another machine is permitted by the network interconnecting the machines. However, because the read and/or write memory access to memory physically located on another computer require the use of the slow network 14, in these configurations such memory accesses can result in substantial delays in memory read/write processing operation, potentially of the order of $10^6 - 10^7$ cycles of the central processing unit of the machine, but ultimately being dependant upon numerous factors, such as for example, the speed, bandwidth, and/or latency of the network 14. This in large part accounts for the diminished performance of the multiple interconnected machines in the prior art arrangement of FIG. 3.

However, in the present arrangement as described above in connection with FIG. 8, it will be appreciated that all reading of memory locations or data is satisfied locally because a current value of all (or some subset of all) memory locations is stored on the machine carrying out the processing which generates the demand to read memory.

Similarly, in the present arrangement as described above in connection with FIG. 8, it will be appreciated that all writing of memory locations or data may be satisfied locally because a current value of all (or some subset of all) memory locations is stored on the machine carrying out the processing which generates the demand to write to memory.

Such local memory read and write processing operation as performed according to the invention can typically be satisfied within $10^2 - 10^3$ cycles of the central processing unit. Thus, in practice, there is substantially less waiting for memory accesses which involves reads than the arrangement shown and described relative to FIG. 3. Additionally, in practice, there may be less waiting for memory accesses which involve writes than the arrangement shown and described relative to FIG. 3.

It may be appreciated that most application software reads memory frequently but writes to memory relatively infrequently. As a consequence, the rate at which memory is being written or re-written is relatively slow compared to the rate at which memory is being read. Because of this slow demand for writing or re-writing of memory, the memory locations or fields can be continually updated at a relatively low speed via the possibly relatively slow and inexpensive commodity network 53, yet this possibly relatively slow speed is sufficient to meet the application program's demand for writing to memory. The result is that the performance of the FIG. 8 arrangement is superior to that of FIG. 3. It may be appreciated in light of the description provided herein that while a relatively slow network communication link or path 53 may advantageously be used because it provides the desired performance and low cost, the invention is not limited to a relatively low speed network connection and may be used with any communication link or path. The invention is transport, network, and communications path independent, and does not depend on how the communication between machines or DRTs takes place. In one embodiment, even electronic mail (email) exchanges between machines or DRTs may suffice for the communications.

In a further optional modification in relation to the above, the identity and changed value pair of a manipulated memory location sent over network 53, each pair typically sent as the sole contents of a single packet, frame or cell for example, can be grouped into batches of multiple pairs of identities and changed values corresponding to multiple manipulated memory locations, and sent together over network 53 or other communications link or path in a single packet, frame, or cell. This further modification further reduces the demands on the communication speed of the network 53 or other communications link or path interconnecting the various machines, as each packet, cell or frame may contain multiple identity and changed value pairs, and therefore fewer packets, frames, or cells require to be sent.

It may be apparent that in an environment where the application program code writes repeatedly to a single memory location, the embodiment illustrated of FIG. 11 of step 114 sends an updating and propagation message to all machines corresponding to every performed memory manipulation operation. In a still further optimal modification in relation to the above, the DRT thread 121/1 of FIG. 12 does not need to perform an updating and propagation operation corresponding to every local memory manipulation operation, but instead may send fewer updating and propagation messages than memory manipulation operations, each message containing the last or latest changed value or content of the manipulated memory location, or optionally may only send a single updating and propagation message corresponding to the last memory manipulation operation. This further improvement reduces the demands on the network 53 or other communications link or path, as fewer packets, frames, or cells require to be sent.

It will also be apparent to those skilled in the art in light of the detailed description provided herein that in a table or list or other data structure created by each DRT 71 when initially recording or creating the list of all, or some subset of all, memory locations (or fields), for each such recorded memory location on each machine M1, ..., Mn there is a name or identity which is common or similar on each of the machines M1, ..., Mn. However, in the individual machines the local memory location corresponding to a given name or identity (listed for example, during step 91 of FIG. 9) will or may vary over time since each machine may and generally will store changed memory values or contents at different memory locations according to its own internal processes. Thus the table, or list, or other data structure in each of the DRTs will have, in general, different local memory locations corresponding to a single memory name or identity, but each global "memory name" or identity will have the same "memory value" stored in the different local memory locations.

It will also be apparent to those skilled in the art in light of the description provided herein that the abovementioned modification of the application program code 50 during loading can be accomplished in many ways or by a variety of means. These ways or means include, but are not limited to at least the following five ways and variations or combinations of these five, including by:

(i) re-compilation at loading,
(ii) by a pre-compilation procedure prior to loading,
(iii) compilation prior to loading,
(iv) a "just-in-time" compilation, or
(v) re-compilation after loading (but, or for example, before execution of the relevant or corresponding application code in a distributed environment).

Traditionally the term "compilation" implies a change in code or language, for example, from source to object code or one language to another. Clearly the use of the term "compilation" (and its grammatical equivalents) in the present specification is not so restricted and can also include or embrace modifications within the same code or language.

Given the fundamental concept of modifying memory manipulation operations to coordinate operation between and amongst a plurality of machines M1....Mn, there are several different ways or embodiments in which this coordinated, coherent and consistent memory state and manipulation operation concept, method, and procedure may be carried out or implemented.

In the first embodiment, a particular machine, say machine M2, loads the asset (such as class or object) inclusive of memory manipulation operation(s), modifies it, and then loads each of the other machines M1, M3, ..., Mn (either sequentially or simultaneously or according to any other order, routine or procedure) with the modified object (or class or other asset or resource) inclusive of the new modified memory manipulation operation. Note that there may be one or a plurality of memory manipulation operations corresponding to only one object in the application code, or there may be a plurality of memory manipulation operations corresponding to a plurality of objects in the application code. Note that in one embodiment, the memory manipulation operation(s) that is (are) loaded is binary executable object code. Alternatively, the memory manipulation operation(s) that is (are) loaded is executable intermediary code.

In this arrangement, which may be termed "master/slave" each of the slave (or secondary) machines M1, M3, ..., Mn loads the modified object (or class), and inclusive of the new modified memory manipulation operation(s), that was sent to it over the computer communications network or other communications link or path by the master (or primary) machine, such as machine M2, or some other machine such as a machine X of FIG. 15. In a slight variation of this "master/slave" or "primary/secondary" arrangement, the computer communications network can be replaced by a shared storage device such as a shared file system, or a shared document/file repository such as a shared database.

Note that the modification performed on each machine or computer need not and frequently will not be the same or identical. What is required is that they are modified in a similar enough way that in accordance with the inventive principles described herein, each of the plurality of machines behaves consistently and coherently relative to the other machines to accomplish the operations and objectives described herein. Furthermore, it will be appreciated in light of the description provided herein that there are a myriad of ways to implement the modifications that may for example depend on the particular hardware, architecture, operating system, application program code, or the like or different factors. It will also be appreciated that embodiments of the invention may be implemented within an operating system, outside of or without the benefit of any operating system, inside the virtual machine, in an EPROM, in software, in firmware, or in any combination of these.

In a still further embodiment, each machine M1, ..., Mn receives the unmodified asset (such as class or object) inclusive of one or more memory manipulation operation(s), but modifies the operations and then loads the asset (such as class or object) consisting of the now modified operations. Although one machine, such as the master or primary machine may customize or perform a different modification to the memory manipulation operation(s) sent to each machine, this embodiment more readily enables the modification carried out by each machine to be slightly different and to be enhanced, customized, and/or optimized based upon its particular machine architecture, hardware, processor, memory, configuration, operating system, or other factors, yet still similar, coherent and consistent with other machines with all other similar modifications and characteristics that may not need to be similar or identical.

In all of the described instances or embodiments, the supply or the communication of the asset code (such as class code or object code) to the machines M1, ..., Mn, and optionally inclusive of a machine X of FIG. 15, can be branched, distributed or communicated among and between the different machines in any combination or permutation; such as by providing direct machine to machine communication (for example, M2 supplies each of M1, M3, M4, etc. directly), or by providing or using cascaded or sequential communication (for example, M2 supplies M1 which then supplies M3 which then supplies M4, and so on), or a combination of the direct and cascaded and/or sequential.

Reference is made to the accompanying Annexure A in which: Annexure A5 is a typical code fragment from a memory manipulation operation prior to modification (e.g., an exemplary unmodified routine with a memory manipulation operation), and Annexure A6 is the same routine with a memory manipulation operation after modification (e.g., an exemplary modified routine with a memory manipulation operation). These code fragments are exemplary only and identify one software code means for performing the modification in an exemplary language. It will be appreciated that other software/firmware or computer program code may be used to accomplish the same or analogous function or operation without departing from the invention.

Annexures A5 and A6 (also reproduced in part in Table VI and Table VII below) are exemplary code listings that set forth the conventional or unmodified computer program software code (such as may be used in a single machine or computer environment) of a routine with a memory manipulation operation of application program code 50 and a post-modification excerpt of the same routine such as may be used in embodiments of the present invention having multiple machines. The modified code that is added to the routine is highlighted in **bold** text.

Table I. Summary Listing of Contents of Annexure A

Annexure A includes exemplary program listings in the JAVA language to further illustrate features, aspects, methods, and procedures of described in the detailed description
A1. This first excerpt is part of an illustration of the modification code of the modifier 51 in accordance with steps 92 and 103 of FIG. 10. It searches through the code array of the application program code 50, and when it detects a memory manipulation instruction (i.e. a putstatic instruction (opcode 178) in the JAVA language and virtual machine environment) it modifies the application program code by the insertion of an "alert" routine. A2. This second excerpt is part of the DRT.alert() method and implements the step of 125 and arrow of 127 of FIG. 12. This DRT.alert() method requests one or more threads of the DRT processing environment of FIG. 12 to update and propagate the value and identity of the changed memory location corresponding to the operation of Annexure A1. A3. This third excerpt is part of the DRT 71, and corresponds to step 128 of FIG. 12. This code fragment shows the DRT in a separate thread, such as thread 121/1 of FIG. 12, after being notified or requested by step 125 and array 127, and sending the changed value and changed value location/identity across the network 53 to the other of the plurality of machines M1...Mn. A4. The fourth excerpt is part of the DRT 71, and corresponds to steps 135 and 136 of FIG. 13. This is a fragment of code to receive a propagated identity and value pair sent by another DRT 71 over the network, and write the changed value to the identified memory location. A5. The fifth excerpt is an disassembled compiled form of the example.java application of Annexure A7, which performs a memory manipulation operation (putstatic and putfield).

Annexure A includes exemplary program listings in the JAVA language to further illustrate features, aspects, methods, and procedures of described in the detailed description

A6. The sixth excerpt is the disassembled compiled form of the same example application in Annexure A5 after modification has been performed by FieldLoader.java of Annexure A11, in accordance with FIG. 9 of this invention. The modifications are highlighted in bold.

A7. The seventh excerpt is the source-code of the example.java application used in excerpt A5 and A6. This example application has two memory locations (staticValue and instanceValue) and performs two memory manipulation operations.

A8. The eighth excerpt is the source-code of FieldAlert.java which corresponds to step 125 and arrow 127 of FIG. 12, and which requests a thread 121/1 executing FieldSend.java of the "distributed run-time" 71 to propagate a changed value and identity pair to the other machines M1...Mn.

A9. The ninth excerpt is the source-code of FieldSend.java which corresponds to step 128 of FIG. 12, and waits for a request/notification generated by FieldAlert.java of A8 corresponding to step 125 and arrow 127, and which propagates a changed value/identity pair requested of it by FieldAlert.java, via network 53.

A10. The tenth excerpt is the source-code of FieldReceive.java, which corresponds to steps 135 and 136 of FIG. 13, and which receives a propagated changed value and identity pair sent to it over the network 53 via FieldSend.java of annexure A9.

A11. FieldLoader.java. This excerpt is the source-code of FieldLoader.java, which modifies an application program code, such as the example.java application code of Annexure A7, as it is being loaded into a JAVA virtual machine in accordance with steps 90, 91, 92, 103, and 94 of FIG. 10. FieldLoader.java makes use of the convenience classes of Annexures A12 through to A36 during the modification of a compiled JAVA

A12. Attribute_info.java
Convience class for representing attribute_info structures within ClassFiles.

A13. ClassFile.java
Convience class for representing ClassFile structures.

A14. Code_attribute.java
Convience class for representing Code_attribute structures within ClassFiles.

A15. CONSTANT_Class_info.java
Convience class for representing CONSTANT_Class_info structures within ClassFiles.

Annexure A includes exemplary program listings in the JAVA language to further illustrate features, aspects, methods, and procedures of described in the detailed description

A16. CONSTANT_Double_info.java
Convience class for representing CONSTANT_Double_info structures within ClassFiles.

A17. CONSTANT_Fieldref_info.java
Convience class for representing CONSTANT_Fieldref_info structures within ClassFiles.

A18. CONSTANT_Float_info.java
Convience class for representing CONSTANT_Float_info structures within ClassFiles.

A19. CONSTANT_Integer_info.java
Convience class for representing CONSTANT_Integer_info structures within ClassFiles.

A20. CONSTANT_InterfaceMethodref_info.java
Convience class for representing CONSTANT_InterfaceMethodref_info structures within ClassFiles.

A21. CONSTANT_Long_info.java
Convience class for representing CONSTANT_Long_info structures within ClassFiles.

A22. CONSTANT_Methodref_info.java
Convience class for representing CONSTANT_Methodref_info structures within ClassFiles.

A23. CONSTANT_NameAndType_info.java
Convience class for representing CONSTANT_NameAndType_info structures within ClassFiles.

A24. CONSTANT_String_info.java
Convience class for representing CONSTANT_String_info structures within ClassFiles.

A25. CONSTANT_Utf8_info.java
Convience class for representing CONSTANT_Utf8_info structures within ClassFiles.

A26. ConstantValue_attribute.java
Convience class for representing ConstantValue_attribute structures within ClassFiles.

A27. cp_info.java
Convience class for representing cp_info structures within ClassFiles.

Annexure A includes exemplary program listings in the JAVA language to further illustrate features, aspects, methods, and procedures of described in the detailed description

A28. Deprecated_attribute.java
Convience class for representing Deprecated_attribute structures within ClassFiles.

A29. Exceptions_attribute.java
Convience class for representing Exceptions_attribute structures within ClassFiles.

A30. field_info.java
Convience class for representing field_info structures within ClassFiles.

A31. InnerClasses_attribute.java
Convience class for representing InnerClasses_attribute structures within ClassFiles.

A32. LineNumberTable_attribute.java
Convience class for representing LineNumberTable_attribute structures within ClassFiles.

A33. LocalVariableTable_attribute.java
Convience class for representing LocalVariableTable_attribute structures within ClassFiles.

A34. method_info.java
Convience class for representing method_info structures within ClassFiles.

A35. SourceFile_attribute.java
Convience class for representing SourceFile_attribute structures within ClassFiles.

A36. Synthetic_attribute.java
Convience class for representing Synthetic_attribute structures within ClassFiles.

Table II. Exemplary code listing showing embodiment of modified code.

A1. This first excerpt is part of an illustration of the modification code of the modifier 51 in accordance with steps 92 and 103 of FIG. 10. It searches through the code array of the application program code 50, and when it detects a memory manipulation instruction (i.e. a putstatic instruction (opcode 178) in the JAVA language and virtual machine environment) it modifies the application program code by the insertion of an "alert" routine.

```
// START
byte[] code =  Code_attribute.code;      // Bytecode of a given method in a
                                         // given classfile.

int code_length = Code_attribute.code_length;

int DRT = 99;        // Location of the CONSTANT_Methodref_info for the
```

```
                    // DRT.alert() method.

for (int i=0; i<code_length; i++){

   if ((code[i] & 0xff) == 179){ // Putstatic instruction.

      System.arraycopy(code, i+3, code, i+6, code_length-(i+3));

      code[i+3] = (byte) 184;      // Invokestatic instruction for the
                                   // DRT.alert() method.

      code[i+4] = (byte) ((DRT >>> 8) & 0xff);

      code[i+5] = (byte) (DRT & 0xff);

   }

}
// END
```

Table III. Exemplary code listing showing embodiment of code for alert method

A2. This second excerpt is part of the DRT.alert() method and implements the step of 125 and arrow of 127 of FIG. 12. This DRT.alert() method requests one or more threads of the DRT processing environment of FIG. 12 to update and propagate the value and identity of the changed memory location corresponding to the operation of Annexure A1.

```
// START
public static void alert(){

   synchronized (ALERT_LOCK){

      ALERT_LOCK.notify(); // Alerts a waiting DRT thread in the background.

   }

}
// END
```

Table IV. Exemplary code listing showing embodiment of code for DRT

A3. This third excerpt is part of the DRT 71, and corresponds to step 128 of FIG. 12. This code fragment shows the DRT in a separate thread, such as thread 121/1 of FIG. 12, after being notified or requested by step 125 and array 127, and sending the changed value and changed value location/identity across the network 53 to the other of the plurality of machines M1...Mn.

```
// START
MulticastSocket ms = DRT.getMulticastSocket();        // The multicast socket
                                                      // used by the DRT for
                                                      // communication.

byte nameTag = 33;          // This is the "name tag" on the network for this
                            // field.

Field field = modifiedClass.getDeclaredField("myField1");    // Stores
                                                             // the field
                                                             // from the
```

```
                                                                 // modified
                                                                 // class.

// In this example, the field is a byte field.
while (DRT.isRunning()){

   synchronized (ALERT_LOCK){

      ALERT_LOCK.wait();    // The DRT thread is waiting for the alert
                            // method to be called.

      byte[] b = new byte[](nameTag, field.getByte(null));   // Stores
                                                              // the
                                                              // nameTag
                                                              // and the
                                                              // value
                                                              // of the
                                                              // field from
                                                              // the modified
                                                              // class in a
                                                              // buffer.

      DatagramPacket dp = new DatagramPacket(b, 0, b.length);

      ms.send(dp);   // Send the buffer out across the network.

   }

}
// END
```

Table V. Exemplary code listing showing embodiment of code for DRT receiving.

A4. The fourth excerpt is part of the DRT 71, and corresponds to steps 135 and 136 of FIG. 13. This is a fragment of code to receive a propagated identity and value pair sent by another DRT 71 over the network, and write the changed value to the identified memory location.

```
// START
MulticastSocket ms = DRT.getMulticastSocket(); // The multicast socket
                                               // used by the DRT for
                                               // communication.

DatagramPacket dp = new DatagramPacket(new byte[2], 0, 2);

byte nameTag = 33;          // This is the "name tag" on the network for this
                            // field.

Field field = modifiedClass.getDeclaredField("myField1");    // Stores the
                                                              // field from
                                                              // the modified
                                                              // class.

// In this example, the field is a byte field.
while (DRT.isRunning){

   ms.receive(dp);  // Receive the previously sent buffer from the network.

   byte[] b = dp.getData();

   if (b[0] == nameTag){   // Check the nametags match.
```

```
        field.setByte(null, b[1]);  // Write the value from the network packet
                                    // into the field location in memory.

    }

}
// END
```

Table VI. Exemplary code listing showing embodiment of application before modification is made.

A5. The fifth excerpt is an disassembled compiled form of the example.java application of Annexure A7, which performs a memory manipulation operation (putstatic and putfield).

```
Method void setValues(int, int)
  0 iload_1
  1 putstatic #3 <Field int staticValue>
  4 aload_0
  5 iload_2
  6 putfield #2 <Field int instanceValue>
  9 return
```

Table VII. Exemplary code listing showing embodiment of application after modification is made.

A6. The sixth excerpt is the disassembled compiled form of the same example application in Annexure A5 after modification has been performed by FieldLoader.java of Annexure A11, in accordance with FIG. 9 of this invention. The modifications are highlighted in **bold**.

```
Method void setValues(int, int)
  0 iload_1
  1 putstatic #3 <Field int staticValue>
  4 ldc #4 <String "example">
  6 iconst_0
  7 invokestatic #5 <Method void alert(java.lang.Object, int)>
  10 aload_0
  11 iload_2
  12 putfield #2 <Field int instanceValue>
  15 aload_0
  16 iconst_1
  17 invokestatic #5 <Method void alert(java.lang.Object, int)>
  20 return
```

Table VIII. Exemplary code listing showing embodiment of source-code of the example application.

A7. The seventh excerpt is the source-code of the example.java application used in excerpt A5 and A6. This example application has two memory locations (staticValue and instanceValue) and performs two memory manipulation operations.

```
import java.lang.*;
```

```
public class example(

   /** Shared static field. */
   public static int staticValue = 0;

   /** Shared instance field. */
   public int instanceValue = 0;

   /** Example method that writes to memory (instance field). */
   public void setValues(int a, int b)(

      staticValue = a;

      instanceValue = b;

   )

)
```

Table IX. Exemplary code listing showing embodiment of the source-code of FieldAlert.

A8. The eighth excerpt is the source-code of FieldAlert.java which corresponds to step 125 and arrow 127 of FIG. 12, and which requests a thread 121/1 executing FieldSend.java of the "distributed run-time" 71 to propagate a changed value and identity pair to the other machines M1...Mn.

```
import java.lang.*;
import java.util.*;
import java.net.*;
import java.io.*;
public class FieldAlert(

   /** Table of alerts. */
   public final static Hashtable alerts = new Hashtable();

   /** Object handle. */
   public Object reference = null;


   /** Table of field alerts for this object. */
   public boolean[] fieldAlerts = null;

   /** Constructor. */
   public FieldAlert(Object o, int initialFieldCount)(
      reference = o;
      fieldAlerts = new boolean[initialFieldCount];
   )

   /** Called when an application modifies a value. (Both objects and
       classes) */
   public static void alert(Object o, int fieldID)(

      // Lock the alerts table.
      synchronized (alerts)(

         FieldAlert alert = (FieldAlert) alerts.get(o);

         if (alert == null)(      // This object hasn't been alerted already,
                                  // so add to alerts table.
            alert = new FieldAlert(o, fieldID + 1);
            alerts.put(o, alert);
         }

         if (fieldID >= alert.fieldAlerts.length){
            // Ok, enlarge fieldAlerts array.
            boolean[] b = new boolean[fieldID+1];
            System.arraycopy(alert.fieldAlerts, 0, b, 0,
               alert.fieldAlerts.length);
            alert.fieldAlerts = b;
         }

         // Record the alert.
         alert.fieldAlerts[fieldID] = true;

         // Mark as pending.
         FieldSend.pending = true;      // Signal that there is one or more
                                        // propagations waiting.

         // Finally, notify the waiting FieldSend thread(s)
         if (FieldSend.waiting){
            FieldSend.waiting = false;
            alerts.notify();
         }

      }

   }

}
```

It is noted that the compiled code in the annexure and portion repeated in the table is taken from the source-code of the file "example.java" which is included in the Annexure A7 (Table VIII). In the procedure of Annexure A5 and Table VI, the procedure name "Method void setValues(int, int)" of Step 001 is the name of the displayed disassembled output of the setValues method of the compiled application code of "example.java". The name "Method void setValues(int, int)" is arbitrary and selected for this example to indicate a typical JAVA method inclusive of a memory manipulation operation. Overall the method is responsible for writing two values to two different memory locations through the use of a memory manipulation assignment statement (being "putstatic" and "putfield" in this example) and the steps to accomplish this are described in turn.

First (Step 002), the Java Virtual Machine instruction "iload_1" causes the Java Virtual Machine to load the integer value in the local variable array at index 1 of the current method frame and store this item on the top of the stack of the current method frame and results in the integer value passed to this method as the first argument and stored in the local variable array at index 1 being pushed onto the stack.

The Java Virtual Machine instruction "putstatic #3 <Field int staticValue>" (Step 003) causes the Java Virtual Machine to pop the topmost value off the stack of the current method frame and store the value in the static field indicated by the CONSTANT_Fieldref_info constant-pool item stored in the 3rd index of the classfile structure of the application program containing this example setValues() method and memory manipulation operation (that is, the "putstatic #3" instruction) of the setValues() method.

Likewise, in this modified setValues() method code, an "aload_0" instruction is inserted after the "putfield #2" instruction in order to be the first instruction following the execution of the "putfield #2" instruction. This causes the JAVA virtual machine to load the instance object of the example class to which the manipulated field of the preceding "putfield #2" instruction belongs, onto the stack of the current method frame and results in the object reference corresponding to the instance field written to by the "putfield #2" instruction, loaded onto the top of the stack of the current method frame. This change is significant because it modifies the setValues() method to load a reference to the object corresponding to the manipulated field onto the stack..

Furthermore, the JAVA virtual machine instruction "iconst_1" is inserted after the "aload_0" instruction so that the JAVA virtual machine loads an integer value of "1" onto the stack of the current method frame and results in the integer value of "1" loaded onto the top of the stack of the current method frame. This change is significant because it modifies the setValues() method to load an integer value, which in this example is "1", which represents the identity of the memory location (field) manipulated by the preceding "putfield #2" operation. It is to be noted that the choice or particular form of the identifier used for the implementation of this invention is for illustration purposes only. In this example, the integer value of "1" corresponds to the "instanceValue" field as the second field of the "example.java" application, as shown in Annexure A7. Therefore, corresponding to the "putfield #2" instruction, the "iconst_1" instruction loads the integer value "1" corresponding to the index of the manipulated field of the "putfield #2" instruction, and which in this case is the second field of "example.java" hence the "1" integer index value, onto the stack.

Additionally, the JAVA virtual machine instruction "invokestatic #5 <Method boolean alert(java.lang.Object, int)>" is inserted after the "iconst_1" instruction so that the JAVA virtual machine pops the two topmost item off the stack of the current method frame (which in accordance with the preceding "aload_0".instruction is a reference to the object corresponding to the object to which the manipulated instance field belongs, and the integer "1" corresponding to the index of the manipulated field in the example.java application) and invokes the "alert" method, passing the two topmost items popped off the stack to the new method frame as its first two arguments. This change is significant because it modifies the setValues() method to execute the "alert" method and associated operations, corresponding to the preceding memory manipulation operation (that is, the "putfield #2" instruction) of the setValues() method.

The method void alert(java.lang.Object, int), part of the FieldAlert code of Annexure A8 and part of the distributed runtime system (DRT) 71, requests or otherwise notifies a DRT thread 121/1 executing the FieldSend.java code of Annexure A9 to update and propagate the changed identity and value of the manipulated memory location to the plurality of machines M1...Mn.

It will be appreciated that the modified code permits, in a distributed computing environment having a plurality of computers or computing machines, the coordinated operation of memory manipulation operations so that the problems associated with the operation of the unmodified code or procedure on a plurality of machines M1...Mn (such as for example inconsistent and incoherent memory state and manipulation and updating operation) does not occur when applying the modified code or procedure.

INITIALIZATION

Returning again to FIG. 14, there is illustrated a schematic representation of a single prior art computer operated as a JAVA virtual machine. In this way, a machine (produced by any one of various manufacturers and having an operating system operating in any one of various different languages) can operate in the particular language of the application program code 50, in this instance the JAVA language. That is, a JAVA virtual machine 72 is able to operate application code 50 in the JAVA language, and utilize the JAVA architecture irrespective of the machine manufacturer and the internal details of the machine.

When implemented in a non-JAVA language or application code environment, the generalized platform, and/or virtual machine and/or machine and/or runtime system is able to operate application code 50 in the language(s) (possibly including for example, but not limited to any one or more of source-code languages, intermediate-code languages, object-code languages, machine-code languages, and any other code languages) of that platform, and/or virtual machine and/or machine and/or runtime system environment, and utilize the platform, and/or virtual machine and/or machine and/or runtime system and/or language architecture irrespective of the machine manufacturer and the internal details of the machine. It will also be appreciated in light of the description provided herein that the platform and/or runtime system may include virtual machine and non-virtual machine software and/or firmware architectures, as well as hardware and direct hardware coded applications and implementations.

Returning to the example of the JAVA language virtual machine environment, in the JAVA language, the class initialization routine <clinit> happens only once when a given class file 50A is loaded. However, the object initialization routine <init> typically happens frequently, for example the object initialization routine may usually occur every time a new object (such as an object 50X, 50Y or 50Z) is created. In addition, within the JAVA environment and other machine or other runtime system environments using classes and object constructs, classes (generally being a broader category than objects) are loaded prior to objects (which are the narrower category and wherein the objects belong to or are identified with a particular class) so that in the application code 50 illustrated in FIG. 14, having a single class 50A and three objects 50X, 50Y, and 50Z, the first class 50A is loaded first, then first object 50X is loaded, then second object 50Y is loaded and finally third object 50Z is loaded.

Where, as in the embodiment illustrated relative to FIG. 14, there is only a single computer or machine 72 (and not a plurality of connected or coupled computers or machines), then no conflict or inconsistency arises in the running of the initialization routines (such as class and object initialization routines) intended to operate during the loading procedure because for conventional operation each initialization routine is executed only once by the single virtual machine or machine or runtime system or language environment as needed for each of the one or more classes and one or more machines, as each machine performs memory manipulation only locally and without any attempt to coordinate or update their local memory state and manipulation operation with any other similar memory state on any one or more other machines. Such an arrangement would therefore be susceptible to inconsistent and incoherent memory state amongst machines M1..Mn due to uncoordinated, inconsistent and/or incoherent memory manipulation and updating operation. Therefore it is desirable to overcome this limitation of the prior art arrangement.

In the exemplary code in Table VII (Annexure A6), the code has been modified so that it solves the problem of consistent, coordinated memory manipulation and updating operation for a plurality of machines M1...Mn, that was not solved in the code example from Table VI (Annexure A5). In this modified setValues() method code, an "ldc #4 <String "example">" instruction is inserted after the "putstatic #3" instruction in order to be the first instruction following the execution of the "putstatic #3" instruction. This causes the JAVA virtual machine to load the String value "example" onto the stack of the current method frame and results in the String value of "example" loaded onto the top of the stack of the current method frame. This change is significant because it modifies the setValues() method to load a String identifier corresponding to the classname of the class containing the static field location written to by the "putstatic #3" instruction onto the stack.

Furthermore, the JAVA virtual machine instruction "iconst_0" is inserted after the "ldc #4" instruction so that the JAVA virtual machine loads an integer value of "0" onto the stack of the current method frame and results in the integer value of "0" loaded onto the top of the stack of the current method frame. This change is significant because it modifies the setValues() method to load an integer value, which in this example is "0", which represents the identity of the memory location (field) manipulated by the preceding "putstatic #3" operation. It is to be noted that the choice or particular form of the memory identifier used for the implementation of this invention is for illustration purposes only. In this example, the integer value of "0" is the identifier used of the manipulated memory location, and corresponds to the "staticValue" field as the first field of the "example.java" application, as shown in Annexure A7. Therefore, corresponding to the "putstatic #3" instruction, the "iconst_0" instruction loads the integer value "0" corresponding to the index of the manipulated field of the "putstatic #3" instruction, and which in this case is the first field of "example.java" hence the "0" integer index value, onto the stack.

Additionally, the JAVA virtual machine instruction "invokestatic #5 <Method boolean alert(java.lang.Object, int)>" is inserted after the "iconst_0" instruction so that the JAVA virtual machine pops the two topmost items off the stack of the current method frame (which in accordance with the preceding "ldc #4" instruction is a reference to the String object with the value "example" corresponding to the name of the class to which manipulated field belongs, and the integer "0" corresponding to the index of the manipulated field in the example.java application) and invokes the "alert" method, passing the two topmost items popped off the stack to the new method frame as its first two arguments. This change is significant because it modifies the setValues() method to execute the "alert" method and associated operations, corresponding to the preceding objects belonging to or identified with the classes, or equivalent where the terms classes and object are not used.

For a more general set of virtual machine or abstract machine environments, and for current and future computers and/or computing machines and/or information appliances or processing systems, and that may not utilize or require utilization of either classes and/or objects, the inventive structure, method, and computer program and computer program product are still applicable. Examples of computers and/or computing machines that do not utilize either classes and/or objects include for example, the x86 computer architecture manufactured by Intel Corporation and others, the SPARC computer architecture manufactured by Sun Microsystems, Inc and others, the PowerPC computer architecture manufactured by International Business Machines Corporation and others, and the personal computer products made by Apple Computer, Inc., and others. For these types of computers, computing machines, information appliances, and the virtual machine or virtual computing environments implemented thereon that do not utilize the idea of classes or objects, the terms 'class' and 'object' may be generalized for example to include primitive data types (such as integer data types, floating point data types, long data types, double data types, string data types, character data types and boolean data types), structured data types (such as arrays and records) derived types, or other code or data structures of procedural languages or other languages and environments such as functions, pointers, components, modules, structures, references and unions.

Returning to the example of the JAVA language virtual machine environment, in the JAVA language, the class initialization routine <clinit> happens only once when a given class file 50A is loaded. However, the object initialization routine <init> typically happens frequently, for example the object initialisation routine will occur every time a new object (such as an object 50X, 50Y and 50Z) is created. In addition, within the JAVA environment and other machine or other runtime system environments using classes and object constructures, classes (being the broader category) are loaded prior to objects (which are the narrower category and wherein the objects belong to or are identified with a particular class) so that in the application code 50 illustrated in FIG. 14, having a single class 50A and three objects 50X-50Z, the first class 50A is loaded first, then the first object 50X is loaded, then second object 50Y is loaded and finally third object 50Z is loaded.

Where, as in the embodiment illustrated relative to FIG. 14, there is only a single computer or machine 72 (not a plurality of connected or coupled machines), then no conflict or inconsistency arises in the running of the initialization routines (i.e. the class initialization routine <clinit> and the object initialisation routine <init>) intended to operate during the loading procedure because for conventional operation each initialisation routine is executed only once by the single virtual machine or machine or runtime system or language environment as needed for each of the one or more classes and one or more objects belonging to or identified with the classes.

However, in the arrangement illustrated in FIG. 8, (and also in FIGS. 31-33), a plurality of individual computers or machines M1, M2, ..., Mn are provided, each of which are interconnected via a communications network 53 or other communications link results in the topmost integer value of the stack of the current method frame being stored in the integer field named "staticValue".

The Java Virtual Machine instruction "aload_0" (Step 004) causes the Java Virtual Machine to load the item in the local variable array at index 0 of the current method frame and store this item on the top of the stack of the current method frame and results in the 'this' object reference stored in the local variable array at index 0 being pushed onto the stack.

First (Step 005), the Java Virtual Machine instruction "iload_2" causes the Java Virtual Machine to load the integer value in the local variable array at index 2 of the current method frame and store this item on the top of the stack of the current method frame and results in the integer value passed to this method as the first argument and stored in the local variable array at index 2 being pushed onto the stack.

The Java Virtual Machine instruction "putfield #2 <Field int instanceValue>" (Step 006) causes the Java Virtual Machine to pop the two topmost values off the stack of the current method frame and store the topmost value in the object instance field of the second popped value, indicated by the CONSTANT_Fieldref_info constant-pool item stored in the 2nd index of the classfile structure of the application program containing this example setValues method and results in the integer value on the top of the stack of the current method frame being stored in the instance field named "instanceValue" of the object reference below the integer value on the stack.

Finally, the JAVA virtual machine instruction "return" (Step 007) causes the JAVA virtual machine to cease executing this setValues() method by returning control to the previous method frame and results in termination of execution of this setValues() method.

As a result of these steps operating on a single machine of the conventional configurations in FIG. 1 and FIG. 2, the JAVA virtual machine manipulates (i.e. writes to) the staticValue and instanceValue memory locations, and in executing the setValues() method containing the memory manipulation operation(s) is able to ensure that memory is and remains consistent between multiple threads of a single application instance, and therefore ensure that unwanted behaviour, such as for example inconsistent or incoherent memory between multiple threads of a single application instance (such inconsistent or incoherent memory being for example incorrect or different values or contents with respect to a single memory location) does not occur. Were these steps to be carried out on the plurality of machines of the configurations of FIG. 5 and FIG. 8 by concurrently executing the application program code 50 on each one of the plurality of machines M1..Mn, the memory manipulation operations of each concurrently executing application program occurrence on each one of the machines would be performed without coordination between any other machine(s), such coordination being for example updating of corresponding memory locations on each machine such that they each report a same content or value. Given the desirable result of consistent, coordinated and coherent memory state and manipulation and updating operation across a plurality of a machines, this prior art arrangement would fail to perform such consistent, coherent, and coordinated memory state and manipulation and updating operation across the plurality of and each of which individual computers or machines provided with a modifier 51 (See in FIG. 5) and realised by or in for example the distributed runtime system(DRT) 71 (See FIG. 8) and loaded with a common application code 50. The term common application program is to be understood to mean an application program or application program code written to operate on a single machine, and loaded and/or executed in whole or in part on each one of the plurality of computers or machines Ml, M2...Mn, or optionally on each one of some subset of the plurality of computers or machines Ml, M2...Mn. Put somewhat differently, there is a common application program represented in application code 50, and this single copy or perhaps a plurality of identical copies are modified to generate a modified copy or version of the application program or program code, each copy or instance prepared for execution on the plurality of machines. At the point after they are modified they are common in the sense that they perform similar operations and operate consistently and coherently with each other. It will be appreciated that a plurality of computers, machines, information appliances, or the like implementing the features of the invention may optionally be connected to or coupled with other computers, machines, information appliances, or the like that do not implement the features of the invention.

In some embodiments, some or all of the plurality of individual computers or machines may be contained within a single housing or chassis (such as so-called "blade servers" manufactured by Hewlett-Packard Development Company, Intel Corporation, IBM Corporation and others) or implemented on a single printed circuit board or even within a single chip or chip set.

Essentially the modifier 51 or DRT 71 or other code modifying means is responsible for modifying the application code 50 so that it may execute initialisation routines or other initialization operations, such as for example class and object initialization methods or routines in the JAVA language and virtual machine environment, in a coordinated, coherent, and consistent manner across and between the plurality of individual machines Ml, M2...Mn. It follows therefore that in such a computing environment it is necessary to ensure that the local objects and classes on each of the individual machines Ml, M2...Mn is initialized in a consistent fashion (with respect to the others).

It will be appreciated in light of the description provided herein that there are alternative implementations of the modifier 51 and the distributed run time 71. For example, the modifier 51 may be implemented as a component of or within the distributed run time 71, and therefore the DRT 71 may implement the functions and operations of the modifier 51. Alternatively, the function and operation of the modifier 51 may be implemented outside of the structure, software, firmware, or other means used to implement the DRT 71. In one embodiment, the modifier 51 and DRT 71 are implemented or written in a single piece of computer program code that provides the functions of the DRT and modifier. The modifier function and structure therefore maybe subsumed into the DRT and considered to be an optional component. Independent of how implemented, the modifier function and structure is responsible for modifying the executable code of the application code program, and the distributed run time function and structure is responsible for implementing communications between and among the computers or machines. The communications functionality in one embodiment is implemented via an intermediary protocol layer within the computer program code of the

DRT on each machine. The DRT may for example implement a communications stack in the JAVA language and use the Transmission Control Protocol/Internet Protocol (TCP/IP) to provide for communications or talking between the machines. Exactly how these functions or operations are implemented or divided between structural and/or procedural elements, or between computer program code or data structures within the invention are less important than that they are provided.

In order to ensure consistent class and object (or equivalent) initialisation status and initialisation operation between and amongst machines M1, M2,..., Mn, the application code 50 is analysed or scrutinized by searching through the executable application code 50 in order to detect program steps (such as particular instructions or instruction types) in the application code 50 which define or constitute or otherwise represent an initialization operation or routine (or other similar memory, resource, data, or code initialization routine or operation). In the JAVA language, such program steps may for example comprise or consist of some part of, or all of, a "<init>" or "<clinit>" method of an object or class, and optionally any other code, routine, or method related to a "<init>" or "<clinit>" method, for example by means of a method invocation from the body of the "<init>" of "<clinit>" method to a different method.

This analysis or scrutiny of the application code 50 may take place either prior to loading the application program code 50, or during the application program code 50 loading procedure, or even after the application program code 50 loading procedure. It may be likened to an instrumentation, program transformation, translation, or compilation procedure in that the application code may be instrumented with additional instructions, and/or otherwise modified by meaning-preserving program manipulations, and/or optionally translated from an input code language to a different code language (such as for example from source-code language or intermediate-code language to object-code language or machine-code language), and with the understanding that the term compilation normally or conventionally involves a change in code or language, for example, from source code to object code or from one language to another language. However, in the present instance the term "compilation" (and its grammatical equivalents) is not so restricted and can also include or embrace modifications within the same code or language. For example, the compilation and its equivalents are understood to encompass both ordinary compilation (such as for example by way of illustration but not limitation, from source-code to object-code), and compilation from source-code to source-code, as well as compilation from object-code to object-code, and any altered combinations therein. It is also inclusive of so-called "intermediary-code languages" which are a form of "pseudo object-code".

By way of illustration and not limitation, in one embodiment, the analysis or scrutiny of the application code 50 may take place during the loading of the application program code such as by the operating system reading the application code from the hard disk or other storage device or source and copying it into memory and preparing to begin execution of the application program code. In another embodiment, in a JAVA virtual machine, the analysis or scrutiny may take place during the class loading procedure of the java.lang.ClassLoader loadClass method (e.g., "java.lang.ClassLoader.loadClass()").

Alternatively, the analysis or scrutiny of the application code 50 may take place even after the application program code loading procedure, such as after the operating system has loaded the application code into memory, or optionally even after execution of the application program code has started or commenced, such as for example after the JAVA virtual machine has loaded the application code into the virtual machine via the "java.lang.ClassLoader.loadClass()" method and optionally commenced execution.

As a consequence, of the above described analysis or scrutiny, initialization routines (for example <clinit> class initialisation methods and <init> object initialization methods) are initially looked for, and when found or identified a modifying code is inserted, so as to give rise to a modified initialization routine. This modified routine is adapted and written to initialize the class 50A on one of the machines, for example JVM#1, and tell, notify, or otherwise communicate to all the other machines M2, ..., Mn that such a class 50A exists and optionally its initialized state. There are several different alternative modes wherein this modification and loading can be carried out.

Thus, in one mode, the DRT 71/1 on the loading machine, in this example Java Virtual Machine M1 (JVM#1), asks the DRT's 71/2...71/n of all the other machines M1, ..., Mn if the similar equivalent first class 50A is initialized (i.e. has already been initialized) on any other machine. If the answer to this question is yes (that is, a similar equivalent class 50A has already been initialized on another machine), then the execution of the initialization procedure is aborted, paused, terminated, turned off or otherwise disabled for the class 50A on machine JVM#1. If the answer is no (that is, a similar equivalent class 50A has not already been initialised on another machine), then the initialization operation is continued (or resumed, or started, or commenced and the class 50A is initialized and optionally the consequential changes (such as for example initialized code and data-structures in memory) brought about during that initialization procedure are transferred to each similar equivalent local class on each one of the other machines as indicated by arrows 83 in FIG. 8.

A similar procedure happens on each occasion that an object, say 50X, 50Y or 50Z is to be loaded and initialized. Where the DRT 71/1 of the loading machine, in this example Java Machine M1 (JVM#1), does not discern, as a result of interrogation of the other machines M2...Mn that, a similar equivalent object to the particular object to be initialized on machine M1, say object 50Y, has already been initialised by another machine, then the DRT 71/1 on machine M1 may execute the object initialization routine corresponding to object 50Y, and optionally each of the other machines M2...Mn may load a similar equivalent local object (which may conveniently be termed a peer object) and associated consequential changes (such as for example initialized data, initialized code, and/or initialized system or resources structures) brought about by the execution of the initialization operation on machine M1. However, if the DRT 71/1 of machine M1 determines that a similar equivalent object to the object 50Y in question has already been initialization on another machine of the plurality of machines (say for example machine M2), then the execution by machine M1 of the initialization function, procedure, or routine corresponding to object 50Y is not started or commenced, or is otherwise aborted, terminated, turned off or otherwise disabled, and object 50Y on machine M1 is loaded, and preferably but optionally the consequential changes (such as for example initialized data, initialized code, and/or other initialized system or resource structures) brought about by the execution of the initialization routine by machine M2, is loaded on machine M1 corresponding to object 50Y. Again there are various ways of bringing about the desired result.

Preferably, execution of the initialization routine is allocated to one machine, such as the first machine M1 to load (and optionally seek to initialize) the object or class. The execution of the initialization routine corresponding to the determination that a particular class or object (and any similar equivalent local classes or objects on each of the machines M1...Mn) is not already initialized, is to execute only once with respect to all machines M1...Mn, and preferably by only one machine, on behalf of all machines M1...Mn. Corresponding to, and preferably following, the execution of the initialization routine by one machine (say machine M1), all other machines may then each load a similar equivalent local object (or class) and optionally load the consequential changes (such as for example initialized data, initialized code, and/or other initialized system or resource structures) brought about by the execution of the initialization operation by machine M1.

As seen in FIG. 15 a modification to the general arrangement of FIG. 8 is provided in that machines M1, M2...Mn are as before and run the same application code 50 (or codes) on all machines M1, M2...Mn simultaneously or concurrently. However, the previous arrangement is modified by the provision of a server machine X which is conveniently able to supply housekeeping functions, for example, and especially the initialisation of structures, assets, and resources. Such a server machine X can be a low value commodity computer such as a PC since its computational load is low. As indicated by broken lines in FIG. 15, two server machines X and X+1 can be provided for redundancy purposes to increase the overall reliability of the system. Where two such server machines X and X+1 are provided, they are preferably but optionally operated as redundant machines in a failover arrangement.

It is not necessary to provide a server machine X as its computational load can be distributed over machines M1, M2...Mn. Alternatively, a database operated by one machine (in a master/slave type operation) can be used for the housekeeping function(s).

FIG. 16 shows a preferred general procedure to be followed. After a loading step 161 has been commenced, the instructions to be executed are considered in sequence and all initialization routines are detected as indicated in step 162. In the JAVA language these are the object initialisation methods (e.g. "<init>") and class initialisation methods (e.g. "<clinit>"). Other languages use different terms.

Where an initialization routine is detected in step 162, it is modified in step 163 in order to perform consistent, coordinated, and coherent initialization operation (such as for example initialization of data structures and code structures) across and between the plurality of machines M1,M2...Mn, typically by inserting further instructions into the initialisation routine to, for example, determine if a similar equivalent object or class (or other asset) on machines M1...Mn corresponding to the object or class (or asset) to which this initialisation routine corresponds, has already been initialised, and if so, aborting, pausing, terminating, turning off, or otherwise disabling the execution of this initialization routine (and/or initialization operation(s)), or if not then starting, continuing, or resuming the executing the initialization routine (and/or initialization operation(s)), and optionally instructing the other machines M1...Mn to load a similar equivalent object or class and consequential changes brought about by the execution of the initialization routine. Alternatively, the modifying instructions may be inserted prior to the routine, such as for example prior to the instruction(s) or operation(s) which commence initialization of the corresponding class or object. Once the modification step 163 has been completed the loading procedure continues by loading the modified application code in place of the unmodified application code, as indicated in step 164. Altogether, the initialization routine is to be executed only once, and preferably by only one machine, on behalf of all machines M1...Mn corresponding to the determination by all machines M1...Mn that the particular object or class (i.e. the similar equivalent local object or class on each machine M1...Mn corresponding to the particular object or class to which this initialization routine relates) has not been initialized.

FIG. 17 illustrates a particular form of modification. After commencing the routine in step 171, the structures, assets or resources (in JAVA termed classes or objects) to be initialised are, in step 172, allocated a name or tag (for example a global name or tag) which can be used to identify corresponding similar equivalent local objects on each of the machines M1,..., Mn. This is most conveniently done via a table (or similar data or record structure) maintained by server machine X of Fig 15. This table may also include an initialization status of the similar equivalent classes or object to be initialised. It will be understood that this table or other data structure may store only the initialization status, or it may store other status or information as well.

As indicated in FIG. 17, if steps 173 and 174 determine by means of the communication between machines M1...Mn by DRT 71 that the similar equivalent local objects on each other machine corresponding to the global name or tag is not already initialised (i.e., not initialized on a machine other than the machine carrying out the loading and seeking to perform initialization), then this means that the object or class can be initialised, preferably but optionally in the normal fashion, by starting, commencing, continuing, or resuming the execution of, or otherwise executing, the initialization routine, as indicated in step 176, since it is the first of the plurality of similar equivalent local objects or classes of machines M1...Mn to be initialized.

In one embodiment, the initialization routine is stopped from initiating or commencing or beginning execution; however, in some implementations it is difficult or practically impossible to stop the initialization routine from initiating or beginning or commencing execution. Therefore, in an alternative embodiment, the execution of the initialization routine that has already started or commenced is aborted such that it does not complete or does not complete in its normal manner. This alternative abortion is understood to include an actual abortion, or a suspend, or postpone, or pause of the execution of a initialization routine that has started to execute (regardless of the stage of execution before completion) and therefore to make sure that the initialization routine does not get the chance to execute to completion the initialization of the object (or class or other asset) – and therefore the object (or class or other asset) remains "un-initialized" (i.e., "not initialized").

However or alternatively, if steps 173 and 174 determine that the global name corresponding to the plurality of similar equivalent local objects or classes, each on a one of the plurality of machines M1...Mn, is already initialised on another machine, then this means that the object or class is considered to be initialized on behalf of, and for the purposes of, the plurality of machines M1....Mn. As a consequence, the execution of the initialisation routine is aborted, terminated, turned off, or otherwise disabled, by carrying out step 175.

FIG. 18, illustrative of one embodiment of step 173 of FIG. 17, shows the inquiry made by the loading machine (one of M1, M2...Mn) to the server machine X of FIG. 15, to enquire as to the initialisation status of the plurality of similar equivalent local objects (or classes) corresponding to the global name. The operation of the loading machine is temporarily interrupted as indicated by step 181, and corresponding to step 173 of FIG. 17, until a reply to this preceding request is received from machine X, as indicated by step 182. In step 181 the loading machine sends an inquiry message to machine X to request the initialization status of the object (or class or other asset) to be initialized. Next, the loading machine awaits a reply from machine X corresponding to the inquiry message sent by the proposing machine at step 181, indicated by step 182.

FIG. 19 shows the activity carried out by machine X of FIG. 15 in response to such an initialization enquiry of step 181 of FIG. 18. The initialization status is determined in steps 192 and 193, which determines if a similar equivalent object (or class or other asset) corresponding to the initialization status request of global name, as received at step 191, is initialized on another machine (i.e. a machine other than the enquiring machine 181 from which the initialization status request of step 191 originates), where a table of initialisation states is consulted corresponding to the record for the global name and, if the initialisation status record indicates that a similar equivalent local object (or class) on another machine (such as on a one of the machines M1...Mn) and corresponding to global name is already initialised, the response to that effect is sent to the enquiring machine by carrying out step 194. Alternatively, if the initialisation status record indicates that a similar equivalent local object (or class) on another machine (such as on a one of the plurality of machines M1...Mn) and corresponding to global name is uninitialized, a corresponding reply is sent to the enquiring machine by carrying out steps 195 and 196. The singular term object or class as used here (or the equivalent term of asset, or resource used in step 192) are to be understood to be inclusive of all similar equivalent objects (or classes, or assets, or resources) corresponding to the same global name on each one of the plurality of machines M1...Mn. The waiting enquiring machine of step 182 is then able to respond and/or operate accordingly, such as for example by (i) aborting (or pausing, or postponing) execution of the initialization routine when the reply from machine X of step 182 indicated that a similar equivalent local object on another machine (such as a one of the plurality of machines M1...Mn) corresponding to the global name of the object proposed to be initialized of step 172 is already initialized elsewhere (i.e. is initialized on a machine other than the machine proposing to carry out the initialization); or (ii) by continuing (or resuming, or starting, or commencing) execution of the initialization routine when the reply from machine X of step 182 indicated that a similar equivalent local object on the plurality of machines M1...Mn corresponding to the global name of the object proposing to be initialized of step 172 is not initialized elsewhere (i.e. not initialized on a machine other than the machine proposing to carry out the initialization).

Reference is made to the accompanying Annexures in which: Annexures A1-A10 illustrate actual code in relation to fields, Annexure B1 is a typical code fragment from an unmodified <clinit> instruction, Annexure B2 is an equivalent in respect of a modified <clinit> instruction, Annexure B3 is a typical code fragment from an unmodified <init> instruction, Annexure B4 is an equivalent in respect of a modified <init> instruction, In addition, Annexure B5 is an alternative to the code of Annexure B2, and Annexure B6 is an alternative to the code of Annexure B4.

Furthermore, Annexure B7 is the source-code of InitClient which carries out one embodiment of the steps of FIGS. 17 and 18, which queries an "initialization server" (for example a machine X) for the initialization status of the specified class or object with respect to the plurality of similar equivalent classes or objects on the plurality of machines M1...Mn. Annexure B8 is the source-code of InitServer which carries out one embodiment of the steps of FIG. 19, which receives an initialization status query sent by InitClient and in response returns the corresponding initialization status of the specified class or object. Similarly, Annexure B9 is the source-code of the example application used in the before/after examples of Annexure B1-B6 (Repeated as Tables X through XV). And, Annexure B10 is the source-code of InitLoader which carries out one embodiment of the steps of FIGS. 16, 20, and 21, which modifies the example application program code of Annexure B9 in accordance with one mode of this invention.

Annexures B1 and B2 (also reproduced in part in Tables X and XI below) are exemplary code listings that set forth the conventional or unmodified computer program software code (such as may be used in a single machine or computer environment) of an initialization routine of application program 50 and a post-modification excerpt of the same initialization routine such as may be used in embodiments of the present invention having multiple machines. The modified code that is added to the initialization routine is highlighted in bold text.

It is noted that the disassembled compiled code in the annexure and portion repeated in the table is taken from the source-code of the file "example.java" which is included in the Annexure B4 (Table XIII). In the procedure of Annexure B1 and Table X, the procedure name "Method <clinit>" of Step 001 is the name of the displayed disassembled output of the clinit method of the compiled application code "example.java". The method name "<clinit>" is the name of a class' initialization method in accordance with the JAVA platform specification, and selected for this example to indicate a typical mode of operation of a JAVA initialization method. Overall the method is responsible for initializing the class 'example' so that it may be used, and the steps the "example.java" code performs are described in turn.

First (Step 002) the JAVA virtual machine instruction "new #2 <Class example>" causes the JAVA virtual machine to instantiate a new class instance of the example class indicated by the CONSTANT_Classref_info constant_pool item stored in the 2nd index of the classfile structure of the application program containing this example <clinit> method and results in a reference to an newly created object of type 'example' being placed (pushed) on the stack of the current method frame of the currently executing thread.

Next (Step 003), the Java Virtual Machine instruction "dup" causes the Java Virtual Machine to duplicate the topmost item of the stack and push the duplicated item onto the topmost position of the stack of the current method frame and results in the reference to the new created 'example' object at the top of the stack being duplicated and pushed onto the stack.

Next (Step 004), the JAVA virtual machine instruction "invokespecial #3 <Method example()>" causes the JAVA virtual machine to pop the topmost item off the stack of the current method frame and invoke the instance initialization method "<init>" on the popped object and results in the "<init>" constructor of the newly created 'example' object being invoked.

The Java Virtual Machine instruction "putstatic #3 <Field example currentExample>" (Step 005) causes the Java Virtual Machine to pop the topmost value off the stack of the current method frame and store the value in the static field indicated by the CONSTANT_Fieldref_info constant-pool item stored in the 3rd index of the classfile structure of the application program containing this example <clinit> method and results in the reference to the newly created and initialized 'example' object on the top of the stack of the current method frame being stored in the static reference field named "currentExample" of class 'example'.

Finally, the Java Virtual Machine instruction "return" (Step 006) causes the Java Virtual Machine to cease executing this <clinit> method by returning control to the previous method frame and results in termination of execution of this <clinit> method.

As a result of these steps operating on a single machine of the conventional configurations in FIG. 1 and FIG. 2, the JAVA virtual machine can keep track of the initialization status of a class in a consistent, coherent and coordinated manner, and in executing the <clinit> method containing the initialization operations is able to ensure that unwanted behaviour (for example execution of the <init> method of class 'example.java' more than once) such as may be caused by inconsistent and/or incoherent initialization operation, does not occur. Were these steps to be carried out on the plurality of machines of the configurations of FIG. 5 and FIG. 8 with the memory update and propagation replication means of FIGS. 9, 10, 11, 12, and 13, and concurrently executing the application program code 50 on each one of the plurality of machines M1...Mn, the initialization operations of each concurrently executing application program occurrence on each one of the machines would be performed without coordination between any other of the occurrences on any other of the machine(s). Given the desirable result of consistent, coordinated and coherent initialization operation across a plurality of a machines, this prior art arrangement would fail to perform such consistent coordinated initialization operation across the plurality of machines, as each machine performs initialization only locally and without any attempt to coordinate their local initialization operation with any other similar initialization operation on any one or more other machines. Such an arrangement would therefore be susceptible to unwanted or other anomalous behaviour due to uncoordinated, inconsistent and/or incoherent initialization states, and associated initialization operation. Therefore it is desirable to overcome this limitation of the prior art arrangement.

In the exemplary code in Table XIV (Annexure B5), the code has been modified so that it solves the problem of consistent, coordinated initialization operation for a plurality of machines M1...Mn, that was not solved in the code example from Table X (Annexure B1). In this modified <clinit> method code, an "ldc #2 <String "example">" instruction is inserted before the "new #5" instruction in order to be the first instruction of the <clinit> method. This causes the JAVA virtual machine to load the item in the constant_pool at index 2 of the current classfile and store this item on the top of the stack of the current method frame, and results in the reference to a String object of value "example" being pushed onto the stack.

Furthermore, the JAVA virtual machine instruction "invokestatic #3 <Method Boolean isAlreadyLoaded(java.lang.String)>" is inserted after the "0 ldc #2" instruction so that the JAVA virtual machine pops the topmost item off the stack of the current method frame (which in accordance with the preceding "ldc #2" instruction is a reference to the String object with the value "example" which corresponds to the name of the class to which this <clinit> method belongs) and invokes the "isAlreadyLoaded" method, passing the popped item to the new method frame as its first argument, and returning a boolean value onto the stack upon return from this "invokestatic" instruction. This change is significant because it modifies the <clinit> method to execute the "isAlreadyLoaded" method and associated operations, corresponding to the start of execution of the <clinit> method, and returns a boolean argument (indicating whether the class corresponding to this <clinit> method is initialized on another machine amongst the plurality of machines M1...Mn) onto the stack of the executing method frame of the <clinit> method.

Next, two JAVA virtual machine instructions "ifeq 9" and "return" are inserted into the code stream after the "2 invokestatic #3" instruction and before the "new #5" instruction. The first of these two instructions, the "ifeq 9" instruction, causes the JAVA virtual machine to pop the topmost item off the stack and performs a comparison between the popped value and zero. If the performed comparison succeeds (i.e. if and only if the popped value is equal to zero), then execution continues at the "9 new #5" instruction. If however the performed comparison fails (i.e. if and only if the popped value is not equal to zero), then execution continues at the next instruction in the code stream, which is the "8 return" instruction. This change is particularly significant because it modifies the <clinit> method to either continue execution of the <clinit> method (i.e. instructions 9-19) if the returned value of the "isAlreadyLoaded" method was negative (i.e. "false"), or discontinue execution of the <clinit> method (i.e. the "8 return" instruction causing a return of control to the invoker of this <clinit> method) if the returned value of the "isAlreadyLoaded" method was positive (i.e. "true").

The method void isAlreadyLoaded(java.lang.String), part of the InitClient code of Annexure B7, and part of the distributed runtime system (DRT) 71, performs the communications operations between machines M1...Mn to coordinate the execution of the <clinit> method amongst the machines M1...Mn. The isAlreadyLoaded method of this example communicates with the InitServer code of Annexure B8 executing on a machine X of FIG. 15, by means of sending an "initialization status request" to machine X corresponding to the class being "initialized" (i.e. the class to which this <clinit> method belongs). With reference to FIG. 19 and Annexure B8, machine X receives the "initialization status request" corresponding to the class to which the <clinit> method belongs, and consults a table of initialization states or records to determine the initialization state for the class to which the request corresponds.

If the class corresponding to the initialization status request is not initialized on another machine other than the requesting machine, then machine X will send a response indicating that the class was not already initialized, and update a record entry corresponding to the specified class to indicate the class is now initialized. Alternatively, if the class corresponding to the initialization status request is initialized on another machine other than the requesting machine, then machine X will send a response indicating that the class is already initialized. Corresponding to the determination that the class to which this initialization status request pertains is not initialized on another machine other than the requesting machine, a reply is generated and sent to the requesting machine indicating that the class is not initialized. Additionally, machine X preferably updates the entry corresponding to the class to which the initialization status request pertained to indicate the class is now initialized. Following a receipt of such a message from machine X indicating that the class is not initialized on another machine, the isAlreadyLoaded() method and operations terminate execution and return a 'false' value to the previous method frame, which is the executing method frame of the <clinit> method. Alternatively, following a receipt of a message from machine X indicating that the class is already initialized on another machine, the isAlreadyLoaded() method and operations terminate execution and return a "true" value to the previous method frame, which is the executing method frame of the <clinit> method. Following this return operation, the execution of the <clinit> method frame then resumes as indicated in the code sequence of Annexure B5 at step 004.

It will be appreciated that the modified code permits, in a distributed computing environment having a plurality of computers or computing machines, the coordinated operation of initialization routines or other initialization operations between and amongst machines M1...Mn so that the problems associated with the operation of the unmodified code or procedure on a plurality of machines M1...Mn (such as for example multiple initialization operation, or re-initialization operation) does not occur when applying the modified code or procedure.

Similarly, the procedure followed to modify an <init> method relating to objects so as to convert from the code fragment of Annexure B3 (See Table XII) to the code fragment of Annexure B6 (See Table XV) is indicated.

Annexures B3 and B6 (also reproduced in part in Tables XII and XV below) are exemplary code listings that set forth the conventional or unmodified computer program software code (such as may be used in a single machine or computer environment) of an initialization routine of application program 50 and a post-modification excerpt of the same initialization routine such as may be used in embodiments of the present invention having multiple machines. The modified code that is added to the initialization routine is highlighted in bold text.

It is noted that the disassembled compiled code in the annexure and portion repeated in the table is taken from the source-code of the file "example.java" which is included in the Annexure B4. In the procedure of Annexure B1 and Table XI, the procedure name "Method <init>" of Step 001 is the name of the displayed disassembled output of the init method of the compiled application code "example.java". The method name "<init>" is the name of an object's initialization method (or methods, as there may be more than one) in accordance with the JAVA platform specification, and selected for this example to indicate a typical mode of operation of a JAVA initialization method. Overall the method is responsible for initializing an 'example' object so that it may be used, and the steps the "example.java" code performs are described in turn.

The Java Virtual Machine instruction "aload_0" (Step 002) causes the Java Virtual Machine to load the item in the local variable array at index 0 of the current method frame and store this item on the top of the stack of the current method frame and results in the 'this' object reference stored in the local variable array at index 0 being pushed onto the stack.

Next (Step 003), the JAVA virtual machine instruction "invokespecial #1 <Method java.lang.Object()>" causes the JAVA virtual machine to pop the topmost item off the stack of the current method frame and invoke the instance initialization method "<init>" on the popped object and results in the "<init>" constructor (or method) of the 'example' object's superclass being invoked.

The Java Virtual Machine instruction "aload_0" (Step 004) causes the Java Virtual Machine to load the item in the local variable array at index 0 of the current method frame and store this item on the top of the stack of the current method frame and results in the 'this' object reference stored in the local variable array at index 0 being pushed onto the stack.

Next (Step 005), the JAVA virtual machine instruction "invokestatic #2 <Method long currentTimeMillis()>" causes the JAVA virtual machine to invoke the "currentTimeMillis()" method of the java.lang.System class, and results in a long value pushed onto the top of the stack corresponding to the return value from the currentTimeMillis() method invocation.

The Java Virtual Machine instruction "putfield #3 <Field long timestamp>" (Step 006) causes the Java Virtual Machine to pop the two topmost values off the stack of the current method frame and store the topmost value in the object instance field of the second popped value, indicated by the CONSTANT_Fieldref_info constant-pool item stored in the 3rd index of the classfile structure of the application program containing this example <init> method, and results in the long value on the top of the stack of the current method frame being stored in the instance field named "timestamp" of the object reference below the long value on the stack.

Finally, the Java Virtual Machine instruction "return" (Step 007) causes the Java Virtual Machine to cease executing this <init> method by returning control to the previous method frame and results in termination of execution of this <init> method.

As a result of these steps operating on a single machine of the conventional configurations in FIG. 1 and FIG. 2, the JAVA virtual machine can keep track of the initialization status of an object in a consistent, coherent and coordinated manner, and in executing the <init> method containing the initialization operations is able to ensure that unwanted behaviour (for example execution of the <init> method of a single 'example.java' object more than once, or re-initialization of the same object) such as may be caused by inconsistent and/or incoherent initialization operation, does not occur. Were these steps to be carried out on the plurality of machines of the configurations of FIG. 5 and FIG. 8 with the memory update and propagation replication means of FIGS. 9, 10, 11, 12, and 13, and concurrently executing the application program code 50 on each one of the plurality of machines M1...Mn, the initialization operations of each concurrently executing application program occurrence on each one of the machines would be performed without coordination between any other of the occurrences on any other of the machine(s). Given the desirable result of consistent, coordinated and coherent initialization operation across a plurality of a machines, this prior art arrangement would fail to perform such consistent coordinated initialization operation across the plurality of machines, as each machine performs initialization only locally and without any attempt to coordinate their local initialization operation with any other similar initialization operation on any one or more other machines. Such an arrangement would therefore be susceptible to unwanted or other anomalous behaviour due to uncoordinated, inconsistent and/or incoherent initialization states, and associated initialization operation. Therefore it is desirable to overcome this limitation of the prior art arrangement.

In the exemplary code in Table XV (Annexure B6), the code has been modified so that it solves the problem of consistent, coordinated initialization operation for a plurality of machines M1...Mn, that was not solved in the code example from Table XII(Annexure B3). In this modified <init> method code, an "aload_0" instruction is inserted after the "1 invokespecial #1" instruction, as the "invokespecial #1" instruction must execute before the object may be further used. This inserted "aload_0" instruction causes the JAVA virtual machine to load the item in the local variable array at index 0 of the current method frame and store this item on the top of the stack of the current method frame, and results in the object reference to the 'this' object at index 0 being pushed onto the stack.

Furthermore, the JAVA virtual machine instruction "invokestatic #3 <Method Boolean isAlreadyLoaded(java.lang.Object)>" is inserted after the "4 aload_0" instruction so that the JAVA virtual machine pops the topmost item off the stack of the current method frame (which in accordance with the preceding "aload_0" instruction is a reference to the object to which this <init> method belongs) and invokes the "isAlreadyLoaded" method, passing the popped item to the new method frame as its first argument, and returning a boolean value onto the stack upon return from this "invokestatic" instruction. This change is significant because it modifies the <init> method to execute the "isAlreadyLoaded" method and associated operations, corresponding to the start of execution of the <init> method, and returns a boolean argument (indicating whether the object corresponding to this <init> method is initialized on another machine amongst the plurality of machines M1...Mn) onto the stack of the executing method frame of the <init> method.

Next, two JAVA virtual machine instructions "ifeq 13" and "return" are inserted into the code stream after the "5 invokestatic #2" instruction and before the "12 aload_0" instruction. The first of these two instructions, the "ifeq 13" instruction, causes the JAVA virtual machine to pop the topmost item off the stack and performs a comparison between the popped value and zero. If the performed comparison succeeds (i.e. if and only if the popped value is equal to zero), then execution continues at the "12 aload_0" instruction. If however the performed comparison fails (i.e. if and only if the popped value is not equal to zero), then execution continues at the next instruction in the code stream, which is the "11 return" instruction. This change is particularly significant because it modifies the <init> method to either continue execution of the <init> method (i.e. instructions 12-19) if the returned value of the "isAlreadyLoaded" method was negative (i.e. "false"), or discontinue execution of the <init> method (i.e. the "11 return" instruction causing a return of control to the invoker of this <init> method) if the returned value of the "isAlreadyLoaded" method was positive (i.e. "true").

The method void isAlreadyLoaded(java.lang.Object), part of the InitClient code of Annexure B7, and part of the distributed runtime system (DRT) 71, performs the communications operations between machines M1...Mn to coordinate the execution of the <init> method amongst the machines M1...Mn. The isAlreadyLoaded method of this example communicates with the InitServer code of Annexure B8 executing on a machine X of FIG. 15, by means of sending an "initialization status request" to machine X corresponding to the object being "initialized" (i.e. the object to which this <clinit> method belongs). With reference to FIG. 19 and Annexure B8, machine X receives the "initialization status request" corresponding to the object to which the <clinit> method belongs, and consults a table of initialization states or records to determine the initialization state for the object to which the request corresponds.

If the object corresponding to the initialization status request is not initialized on another machine other than the requesting machine, then machine X will send a response indicating that the object was not already initialized, and update a record entry corresponding to the specified object to indicate the object is now initialized. Alternatively, if the object corresponding to the initialization status request is initialized on another machine other than the requesting machine, then machine X will send a response indicating that the object is already initialized. Corresponding to the determination that the object to which this initialization status request pertains is not initialized on another machine other than the requesting machine, a reply is generated and sent to the requesting machine indicating that the object is not initialized. Additionally, machine X preferably updates the entry corresponding to the object to which the initialization status request pertained to indicate the object is now initialized. Following a receipt of such a message from machine X indicating that the object is not initialized on another machine, the isAlreadyLoaded() method and operations terminate execution and return a 'false' value to the previous method frame, which is the executing method frame of the <init> method. Alternatively, following a receipt of a message from machine X indicating that the object is already initialized on another machine, the isAlreadyLoaded() method and operations terminate execution and return a "true" value to the previous method frame, which is the executing method frame of the <init> method. Following this return operation, the execution of the <init> method frame then resumes as indicated in the code sequence of Annexure B5 at step 006.

It will be appreciated that the modified code permits, in a distributed computing environment having a plurality of computers or computing machines, the coordinated operation of initialization routines or other initialization operations so that the problems associated with the operation of the unmodified code or procedure on a plurality of machines M1...Mn (such as for example multiple initialization, or re-initialization operation) does not occur when applying the modified code or procedure.

Annexure B1 is a before-modification excerpt of the disassembled compiled form of the <clinit> method of the example.java application of Annexure B9. Annexure B2 is an after-modification form of Annexure B1, modified by InitLoader.java of Annexure B10 in accordance with the steps of FIG. 20. Annexure B3 is a before-modification excerpt of the disassembled compiled form of the <init> method of the example.java application of Annexure B9. Annexure B4 is an after-modification form of Annexure B3, modified by InitLoader.java of Annexure B10 in accordance with the steps of FIG. 21. Annexure B5 is an alternative after-modification form of Annexure B1, modified by InitLoader.java of Annexure B10 in accordance with the steps of FIG. 20. And Annexure B6 is an alternative after-modification form of Annexure B3, modified by InitLoader.java of Annexure B10 in accordance with the steps of FIG. 21. The modifications are highlighted in **bold**.

Table X. Annexure B1

B1
```
Method <clinit>
  0 new #2 <Class example>
  3 dup
  4 invokespecial #3 <Method example()>
  7 putstatic #4 <Field example currentExample>
 10 return
```

Table XI. Annexure B2

B2
```
Method <clinit>
  0 invokestatic #3 <Method boolean isAlreadyLoaded()>
  3 ifeq 7
  6 return
  7 new #5 <Class example>
 10 dup
 11 invokespecial #6 <Method example()>
 14 putstatic #7 <Field example example>
 17 return
```

Table XII. Annexure B3

B3
```
Method <init>
  0 aload_0
```

```
 1 invokespecial #1 <Method java.lang.Object()>
 4 aload_0
 5 invokestatic #2 <Method long currentTimeMillis()>
 8 putfield #3 <Field long timestamp>
11 return
```

Table XIII. Annexure B4

```
B4
Method <init>
  0 aload_0
  1 invokespecial #1 <Method java.lang.Object()>
  4 invokestatic #2 <Method boolean isAlreadyLoaded()>
  7 ifeq 11
 10 return
 11 aload_0
 12 invokestatic #4 <Method long currentTimeMillis()>
 15 putfield #5 <Field long timestamp>
 18 return
```

Table XIV. Annexure B5

```
B5
Method <clinit>
  0 ldc #2 <String "example">
  2 invokestatic #3 <Method boolean isAlreadyLoaded(java.lang.String)>
  5 ifeq 9
  8 return
  9 new #5 <Class example>
 12 dup
 13 invokespecial #6 <Method example()>
 16 putstatic #7 <Field example currentExample>
 19 return
```

Table XV. Annexure B6

```
B6
Method <init>
  0 aload_0
  1 invokespecial #1 <Method java.lang.Object()>
  4 aload_0
  5 invokestatic #2 <Method boolean isAlreadyLoaded(java.lang.Object)>
  8 ifeq 12
 11 return
 12 aload_0
 13 invokestatic #4 <Method long currentTimeMillis()>
 16 putfield #5 <Field long timestamp>
 19 return
```

Turning now to FIGS. 20 and 21, the procedure followed to modify class initialisation routines (i.e., the "<clinit>"method) and object initialization routines (i.e. the "<init>" method) is presented. The procedure followed to modify a <clinit> method relating to classes so as to convert from the code fragment of Annexure B1 (See Table X) to the code fragment of Annexure B5 (See Table XIV) is indicated.

Similarly, the procedure followed to modify an object initialization <init> method relating to objects so as to convert from the code fragment of Annexure B3 (See Table XII) to the code fragment of Annexure B6 (See Table XV) is indicated.

The initial loading of the application code 50 (an illustrative example in source-code form of which is displayed in Annexure B9, and a corresponding partially disassembled form of which is displayed in Annexure B1 (See also Table X) and Annexure B3 (See also Table XII)) onto the JAVA virtual machine 72 is commenced at step 201, and the code is analysed or scrutinized in order to detect one or more class initialization instructions, code-blocks or methods (i.e. "<clinit>" methods) by carrying out step 202, and/or one or more object initialization instructions, code-blocks, or methods (i.e. "<init>" methods) by carrying out step 212. Once so detected, an <clinit> method is modified by carrying out step 203, and an <init> method is modified by carrying out step 213. One example illustration for a modified class initialisation routine is indicated in Annexure B2 (See also Table XI), and a further illustration of which is indicated in Annexure B5 (See also Table XIV). One example illustration for a modified object initialisation routine is indicated in Annexure B4 (See also Table XIII), and a further illustration of which is indicated in Annexure B6 (See also Table XV). As indicated by step 204 and 214, after the modification is completed the loading procedure is then continued such that the modified application code is loaded into or onto each of the machines instead of the unmodified application code.

Annexure B1 (See also Table X) and Annexure B2 (See also Table XI) are the before (or pre-modification or unmodified code) and after (or post-modification or modified code) excerpt of a class initialisation routine (i.e. a "<clinit>" method) respectively. Additionally, a further example of an alternative modified <clinit> method is illustrated in Annexure B5 (See also Table XIV). The modified code that is added to the method is highlighted in bold. In the unmodified partially disassembled code sample of Annexure B1, the "new #2" and "invokespecial #3" instructions of the <clinit> method creates a new object (of the type 'example'), and the following instruction "putstatic #4" writes the reference of this newly created object to the memory location (field) called "currentExample". Thus, without management of coordinated class initialisation in a distributed environment of a plurality of machines M1, ..., Mn, and each with a memory updating and propagation means of FIG. 9, 10, 11, 12, and 13, whereby the application program code 50 is to operate as a single co-ordinated, consistent, and coherent instance across the plurality of machines M1...Mn, each computer or computing machine would re-initialise (and optionally alternatively re-write or over-write) the "currentExample" memory location (field) with multiple and different objects corresponding to the multiple executions of the <clinit> method, leading to potentially incoherent or inconsistent memory between and amongst the occurrences of the application program code 50 on each of the machines M1, ..., Mn. Clearly this is not what the programmer or user of a single application program code 50 instance expects to happen.

So, taking advantage of the DRT, the application code 50 is modified as it is loaded into the machine by changing the class initialisation routine (i.e., the <clinit> method). The changes made (highlighted in bold) are the initial instructions that the modified <clinit> method executes. These added instructions determine the initialization status of this particular class by checking if a similar equivalent local class on another machine corresponding to this particular class, has already been initialized and optionally loaded, by calling a routine or procedure to determine the initialization status of the plurality of similar equivalent classes, such as the "is already loaded" (e.g., "isAlreadyLoaded()") procedure or method. The "isAlreadyLoaded()" method of InitClient of Annexure B7 of DRT 71 performing the steps of 172-176 of FIG. 17 determines the initialization status of the similar equivalent local classes each on a one of the machines M1, ..., Mn corresponding to the particular class being loaded, the result of which is either a true result or a false result corresponding to whether or not another one (or more) of the machines M1...Mn have already initialized, and optionally loaded, a similar equivalent class.

The initialisation determination procedure or method "isAlreadyLoaded()" of InitClient of Annexure B7 of the DRT 71 can optionally take an argument which represents a unique identifier for this class (See Annexure B5 and Table XIV). For example, the name of the class that is being considered for initialisation, a reference to the class or class-object representing this class being considered for initialization, or a unique number or identifier representing this class across all machines (that is, a unique identifier corresponding to the plurality of similar equivalent local classes each on a one of the plurality of machines M1...Mn), to be used in the determination of the initialisation status of the plurality of similar equivalent local classes on each of the machines M1...Mn. This way, the DRT can support the initialization of multiple classes at the same time without becoming confused as to which of the multiple classes are already loaded and which are not, by using the unique identifier of each class.

The DRT 71 can determine the initialization status of the class in a number of possible ways. Preferably, the requesting machine can ask each other requested machine in turn (such as by using a computer communications network to exchange query and response messages between the requesting machine and the requested machine(s)) if the requested machine's similar equivalent local class corresponding to the unique identifier is initialized, and if any requested machine replies true indicating that the similar equivalent local class has already been initialized, then return a true result at return from the isAlreadyLoaded() method indicating that the local class should not be initialized, otherwise return a false result at return from the isAlreadyLoaded() method indicating that the local class should be initialized. Of course different logic schemes for true or false results may alternatively be implemented with the same effect. Alternatively, the DRT on the local machine can consult a shared record table (perhaps on a separate machine (eg machine X), or a coherent shared record table on each local machine and updated to remain substantially identical, or in a database) to determine if one of the plurality of similar equivalent classes on other machines has been initialised.

If the isAlreadyLoaded() method of the DRT 71 returns false, then this means that this class (of the plurality of similar equivalent local classes on the plurality of machines M1...Mn) has not been initialized before on any other machine in the distributed computing environment of the plurality of machines M1...Mn, and hence, the execution of the class initialisation method is to take place or proceed as this is considered the first and original initialization of a class of the plurality of similar equivalent classes on each machine. As a result, when a shared record table of initialisation states exists, the DRT must update the initialisation status record corresponding to this class in the shared record table to true or other value indicating that this class is initialized, such that subsequent consultations of the shared record table of initialisation states (such as performed by all subsequent invocations of isAlreadyLoaded method) by all machines, and optionally including the current machine, will now return a true value indicating that this class is already initialized. Thus, if isAlreadyLoaded() returns false, the modified class initialisation routine resumes or continues (or otherwise optionally begins or starts) execution.

On the other hand, if the isAlreadyLoaded method of the DRT 71 returns true, then this means that this class (of the plurality of similar equivalent local classes each on one of the plurality of machines M1...Mn) has already been initialised in the distributed environment, as recorded in the shared record table on machine X of the initialisation states of classes. In such a case, the class initialisation method is not to be executed (or alternatively resumed, or continued, or started, or executed to completion), as it will potentially cause unwanted interactions or conflicts, such as re-initialization of memory, data structures or other machine resources or devices. Thus, when the DRT returns true, the inserted instructions at the start of the <clinit> method prevent execution of the initialization routine (optionally in whole or in part) by aborting the start or continued execution of the <clinit> method through the use of the return instruction, and consequently aborting the JAVA Virtual Machine's initialization operation for this class.

An equivalent procedure for the initialization routines of object (for example "<init>" methods) is illustrated in FIG. 21 where steps 212 and 213 are equivalent to steps 202 and 203 of FIG. 20. This results in the code of Annexure B3 being converted into the code of Annexure B4 (See also Table XIII) or Annexure B6 (See also Table XV).

Annexure B3 (See also Table XII) and Annexure B4 (See also Table XIV) are the before (or pre-modification or unmodified code) and after (or post-modification or modified code) excerpt of a object initialisation routine (i.e. a "<init>" method) respectively. Additionally, a further example of an alternative modified <init> method is illustrated in Annexure B6 (See also Table XV). The modified code that is added to the method is highlighted in bold. In the unmodified partially disassembled code sample of Annexure B4, the "aload_0" and "invokespecial #3" instructions of the <init> method invokes the <init> of the java.lang.Object superclass. Next, the following instructions "aload_0" loads a reference to the 'this' object onto the stack to be one of the arguments to the "8 putfield #3" instruction. Next, the following instruction "invokestatic #2" invokes the method java.lang.System.currentTimeMillis() and returns a long value on the stack. Next the following instruction "putfield #3" writes the long value placed on the stack be the preceding "invokestatic #2" instruction to the memory location (field) called "timestamp" corresponding to the object instance loaded on the stack by the "4 aload_0" instruction. Thus, without management of coordinated object initialisation in a distributed environment of a plurality of machines M1, ..., Mn, and each with a memory updating and propagation means of FIG. 9, 10, 11, 12, and 13, whereby the application program code 50 is to operate as a single co-ordinated, consistent, and coherent instance across the plurality of machines M1...Mn, each computer or computing machine would re-initialise (and optionally alternatively re-write or over-write) the "timestamp" memory location (field) with multiple and different values corresponding to the multiple executions of the <init> method, leading to potentially incoherent or inconsistent memory between and amongst the occurrences of application program code 50 on each of the machines M1, ..., Mn. Clearly this is not what the programmer or user of a single application program code 50 instance expects to happen.

So, taking advantage of the DRT, the application code 50 is modified as it is loaded into the machine by changing the object initialisation routine (i.e. the <init> method). The changes made (highlighted in bold) are the initial instructions that the modified <init> method executes. These added instructions determine the initialisation status of this particular object by checking if a similar equivalent local object on another machine corresponding to this particular object, has already been initialized and optionally loaded, by calling a routine or procedure to determine the initialisation status of the object to be initialised, such as the "is already loaded" (e.g., "isAlreadyLoaded()") procedure or method of Annexure B7. The "isAlreadyLoaded()" method of DRT 71 performing the steps of 172-176 of FIG. 17 determines the initialization status of the similar equivalent local objects each on a one of the machines M1, ..., Mn corresponding to the particular object being loaded, the result of which is either a true result or a false result corresponding to whether or not another one (or more) of the machines M1...Mn have already initialized, and optionally loaded, this object.

The initialisation determination procedure or method "isAlreadyLoaded()" of the DRT 71 can optionally take an argument which represents a unique identifier for this object (See Annexure B6 and Table XV). For example, the name of the object that is being considered for initialisation, a reference to the object being considered for initialization, or a unique number or identifier representing this object across all machines (that is, a unique identifier corresponding to the plurality of similar equivalent local objects each on a one of the plurality of machines M1...Mn), to be used in the determination of the initialisation status of this object in the plurality of similar equivalent local objects on each of the machines M1...Mn. This way, the DRT can support the initialization of multiple objects at the same time without becoming confused as to which of the multiple objects are already loaded and which are not, by using the unique identifier of each object.

The DRT 71 can determine the initialization status of the object in a number of possible ways. Preferably, the requesting machine can ask each other requested machine in turn (such as by using a computer communications network to exchange query and response messages between the requesting machine and the requested machine(s)) if the requested machine's similar equivalent local object corresponding to the unique identifier is initialized, and if any requested machine replies true indicating that the similar equivalent local object has already been initialized, then return a true result at return from the isAlreadyLoaded() method indicating that the local object should not be initialized, otherwise return a false result at return from the isAlreadyLoaded() method indicating that the local object should be initialized. Of course different logic schemes for true or false results may alternatively be implemented with the same effect. Alternatively, the DRT on the local machine can consult a shared record table (perhaps on a separate machine (eg machine X), or a coherent shared record table on each local machine and updated to remain substantially identical, or in a database) to determine if this particular object (or any one of the plurality of similar equivalent objects on other machines) has been initialised by one of the requested machines.

If the isAlreadyLoaded() method of the DRT 71 returns false, then this means that this object (of the plurality of similar equivalent local objects on the plurality of machines M1...Mn) has not been initialized before on any other machine in the distributed computing environment of the plurality of machines M1...Mn, and hence, the execution of the object initialisation method is to take place or proceed as this is considered the first and original initialization. As a result, when a shared record table of initialisation states exists, the DRT must update the initialisation status record corresponding to this object in the shared record table to true or other value indicating that this object is initialized, such that subsequent consultations of the shared record table of initialisation states (such as performed by all subsequent invocations of isAlreadyLoaded method) by all machines, and including the current machine, will now return a true value indicating that this object is already initialized. Thus, if isAlreadyLoaded() returns false, the modified object initialisation routine resumes or continues (or otherwise optionally begins or starts) execution..

On the other hand, if the isAlreadyLoaded method of the DRT 71 returns true, then this means that this object (of the plurality of similar equivalent local objects each on one of the plurality of machines M1...Mn) has already been initialised in the distributed environment, as recorded in the shared record table on machine X of the initialisation states of objects. In such a case, the object initialisation method is not to be executed (or alternatively resumed, or continued, or started, or executed to completion), as it will potentially cause unwanted interactions or conflicts, such as re-initialization of memory, data structures or other machine resources or devices. Thus, when the DRT returns true, the inserted instructions near the start of the <init> method prevent execution of the initialization routine (optionally in whole or in part) by aborting the start or continued execution of the <init> method through the use of the return instruction, and consequently aborting the JAVA Virtual Machine's initialization operation for this object.

A similar modification as used for <clinit> is used for <init>. The application program's <init> method (or methods, as there may be multiple) is or are detected as shown by step 212 and modified as shown by step 213 to behave coherently across the distributed environment.

The disassembled instruction sequence after modification has taken place is set out in Annexure B4 (and an alternative similar arrangement is provided in Annexure B6) and the modified/inserted instructions are highlighted in bold. For the <init> modification, unlike the <clinit> modification, the modifying instructions are often required to be placed after the "invokespecial" instruction, instead of at the very beginning. The reasons for this are driven by the JAVA Virtual Machine specification. Other languages often have similar subtle design nuances.

Given the fundamental concept of testing to determine if initialization has already been carried out on a one of a plurality of similar equivalent classes or object or other asset each on a one of the machines M1...Mn, and if not carrying out the initialization, and if so, not carrying out the initialization; there are several different ways or embodiments in which this coordinated and coherent initialization concept, method, and procedure may be carried out or implemented.

In the first embodiment, a particular machine, say machine M2, loads the asset (such as class or object) inclusive of an initialisation routine, modifies it, and then loads each of the other machines M1, M3, ..., Mn (either sequentially or simultaneously or according to any other order, routine or procedure) with the modified object (or class or other asset or resource) inclusive of the new modified initialization routine(s). Note that there may be one or a plurality of routines corresponding to only one object in the application code, or there may be a plurality of routines corresponding to a plurality of objects in the application code. Note that in one embodiment, the initialization routine(s) that is (are) loaded is binary executable object code. Alternatively, the initialization routine(s) that is (are) loaded is executable intermediary code.

In this arrangement, which may be termed "master/slave" each of the slave (or secondary) machines M1, M3, ..., Mn loads the modified object (or class), and inclusive of the new modified initialisation routine(s), that was sent to it over the computer communications network or other communications link or path by the master (or primary) machine, such as machine M2, or some other machine such as a machine X of FIG. 15. In a slight variation of this "master/slave" or "primary/secondary" arrangement, the computer communications network can be replaced by a shared storage device such as a shared file system, or a shared document/file repository such as a shared database.

Note that the modification performed on each machine or computer need not and frequently will not be the same or identical. What is required is that they are modified in a similar enough way that in accordance with the inventive principles described herein, each of the plurality of machines behaves consistently and coherently relative to the other machines to accomplish the operations and objectives described herein. Furthermore, it will be appreciated in light of the description provided herein that there are a myriad of ways to implement the modifications that may for example depend on the particular hardware, architecture, operating system, application program code, or the like or different factors. It will also be appreciated that embodiments of the invention may be implemented within an operating system, outside of or without the benefit of any operating system, inside the virtual machine, in an EPROM, in software, in firmware, or in any combination of these.

In a further variation of this "master/slave" or "primary/secondary" arrangement, machine M2 loads asset (such as class or object) inclusive of an (or even one or more) initialization routine in unmodified form on machine M2, and then (for example, machine M2 or each local machine) modifies the class (or object or asset) by deleting the initialization routine in whole or part from the asset (or class or object) and loads by means of a computer communications network or other communications link or path the modified code for the asset with the now modified or deleted initialization routine on the other machines. Thus in this instance the modification is not a transformation, instrumentation, translation or compilation of the asset initialization routine but a deletion of the initialization routine on all machines except one.

The process of deleting the initialization routine in its entirety can either be performed by the "master" machine (such as machine M2 or some other machine such as machine X of FIG. 15) or alternatively by each other machine M1, M3, ..., Mn upon receipt of the unmodified asset. An additional variation of this "master/slave" or "primary/secondary" arrangement is to use a shared storage device such as a shared file system, or a shared document/file repository such as a shared database as means of exchanging the code (including for example, the modified code) for the asset, class or object between machines M1, M2,...,Mn and optionally a machine X of FIG. 15.

In a still further embodiment, each machine M1, ..., Mn receives the unmodified asset (such as class or object) inclusive of one or more initialization routines, but modifies the routines and then loads the asset (such as class or object) consisting of the now modified routines. Although one machine, such as the master or primary machine may customize or perform a different modification to the initialization routine sent to each machine, this embodiment more readily enables the modification carried out by each machine to be slightly different and to be enhanced, customized, and/or optimized based upon its particular machine architecture, hardware, processor, memory, configuration, operating system, or other factors, yet still similar, coherent and consistent with other machines with all other similar modifications and characteristics that may not need to be similar or identical.

In a further arrangement, a particular machine, say M1, loads the unmodified asset (such as class or object) inclusive of one or more initialisation routine and all other machines M2, M3, ..., Mn perform a modification to delete the initialization routine of the asset (such as class or object) and load the modified version.

In all of the described instances or embodiments, the supply or the communication of the asset code (such as class code or object code) to the machines M1, ..., Mn, and optionally inclusive of a machine X of FIG. 15, can be branched, distributed or communicated among and between the different machines in any combination or permutation; such as by providing direct machine to machine communication (for example, M2 supplies each of M1, M3, M4, etc. directly), or by providing or using cascaded or sequential communication (for example, M2 supplies M1 which then supplies M3 which then supplies M4, and so on), or a combination of the direct and cascaded and/or sequential.

In a still further arrangement, the initial machine, say M2, can carry out the initial loading of the application code 50, modify it in accordance with this invention, and then generate a class/object loaded and initialised table which lists all or at least all the pertinent classes and/or objects loaded and initialised by machine M2. This table is then sent or communicated (or at least its contents are sent or communicated) to all other machines (including for example in branched or cascade fashion). Then if a machine, other than M2, needs to load and therefore initialise a class listed in the table, it sends a request to M2 to provide the necessary information, optionally consisting of either the unmodified application code 50 of the class or object to be loaded, or the modified application code of the class or object to be loaded, and optionally a copy of the previously initialised (or optionally and if available, the latest or even the current) values or contents of the previously loaded and initialised class or object on machine M2. An alternative arrangement of this mode may be to send the request for necessary information not to machine M2, but some other, or even more than one of, machine M1, ..., Mn or machine X. Thus the information provided to machine Mn is, in general, different from the initial state loaded and initialise by machine M2.

. Under the above circumstances it is preferable and advantageous for each entry in the table to be accompanied by a counter which is incremented on each occasion that a class or object is loaded and initialised on one of the machines M1, ..., Mn. Thus, when data or other content is demanded, both the class or object contents and the count of the corresponding counter, and optionally in addition the modified or unmodified application code, are transferred in response to the demand. This "on demand" mode may somewhat increase the overhead of the execution of this invention for one or more machines M1, ..., Mn, but it also reduces the volume of traffic on the communications network which interconnects the computers and therefore provides an overall advantage.

In a still further arrangement, the machines M1 to Mn, may send some or all load requests to an additional machine X (see for example the embodiment of FIG. 15), which performs the modification to the application code 50 inclusive of an (and possibly a plurality of) initialisation routine(s) via any of the afore mentioned methods, and returns the modified application code inclusive of the now modified initialization routine(s) to each of the machines M1 to Mn, and these machines in turn load the modified application code inclusive of the modified routines locally. In this arrangement, machines M1 to Mn forward all load requests to machine X, which returns a modified application program code 50 inclusive of modified initialization routine(s) to each machine. The modifications performed by machine X can include any of the modifications covered under the scope of the present invention. This arrangement may of course be applied to some of the machines and other arrangements described herein before applied to other of the machines.

Persons skilled in the computing arts will be aware of various possible techniques that may be used in the modification of computer code, including but not limited to instrumentation, program transformation, translation, or compilation means.

One such technique is to make the modification(s) to the application code, without a preceding or consequential change of the language of the application code. Another such technique is to convert the original code (for example, JAVA language source-code) into an intermediate representation (or intermediate-code language, or pseudo code), such as JAVA byte code. Once this conversion takes place the modification is made to the byte code and then the conversion may be reversed. This gives the desired result of modified JAVA code.

A further possible technique is to convert the application program to machine code, either directly from source-code or via the abovementioned intermediate language